Dive deep into Stuntman's closed-loop automation

Introduction

Most lab automation stops at execution. A fixed protocol may run hands-free, but experimental design, analysis, and decision-making often remain disconnected. A scientist still has to assemble the data, judge the outcome, choose what to try next, and launch another run. Closed-loop automation connects those steps so the result of one experiment can automatically shape the next.

Stuntman (Figure 1) provides the automation needed to turn each new decision into an executed experiment. This technical note goes under the hood of the demonstrated workflow: how configuration files and Python scripts define the loop, how Stuntman AI supports script development, how validated scripts coordinate execution, and how a Gaussian-process Bayesian optimizer selects conditions at runtime.

The closed-loop cycle

A closed-loop workflow repeats four connected stages—design, execute, measure, and learn (Figure 2).

- **Design** translates the conditions selected by the decision engine into an executable Stuntman design.
- **Execute** runs those conditions on Stuntman's modular hardware deck.
- **Measure** captures experimental results from integrated analytical equipment.
- **Learn** converts the measurements into objective values and returns the results to the decision engine.

Figure 1: Stuntman: native AI-driven, fundamentally flexible automation for closed-loop experimentation.

Key characteristics of the closed-loop:

Autonomy: In the demonstrated run, validated scripts coordinated cycles without requiring scientist input between cycles.

Configuration-driven control: XLSX or JSON files separate run settings from workflow logic.

Traceability: Design, execution, and measurement data remain together in the .stunt file.

AI-assisted development: Stuntman AI helped generate and refine the closed-loop workflow scripts before execution.

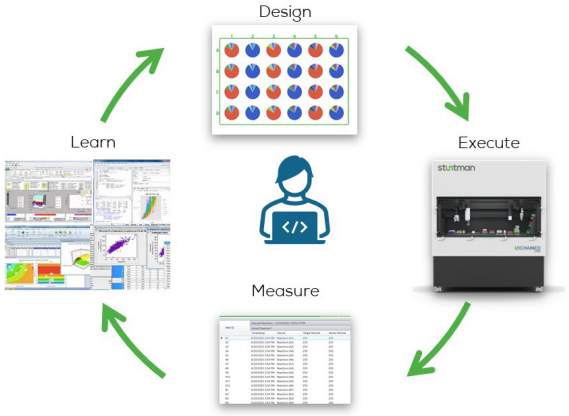

Figure 2: The closed-loop cycle.

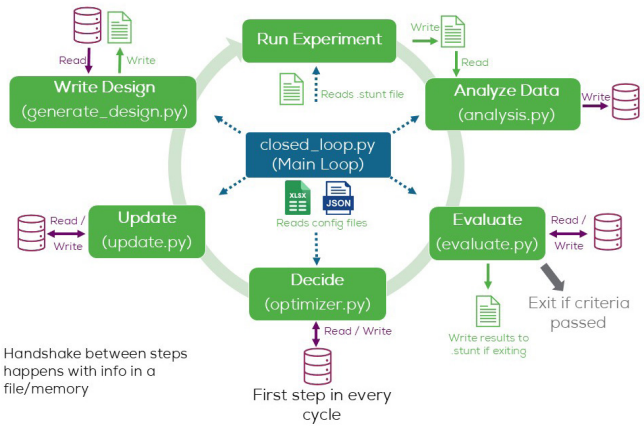

Figure 3: Stuntman's closed-loop workflow architecture.

Workflow architecture

The demonstrated loop uses a lightweight, modular architecture coordinated by closed_loop.py (Figure 3). The main script reads the run configuration, and calls the workflow functions in sequence for optimization, state updates, design generation, execution, analysis, and evaluation.

Each function has one defined responsibility. Separating optimization, design generation, execution, analysis, and evaluation keeps the decision logic independent from the physical experiment while allowing each component to be reviewed or modified without rewriting the entire loop.

Configuration and data model

Experiment-specific inputs remain outside the closed-loop code. The XLSX or JSON configuration defines materials and their roles, experimental settings, variable ranges, optimization criteria, and cycle limits. Scientists can therefore review or adjust the experiment without rewriting the sequence of the loop.

At execution, generate_design.py packages the selected conditions into a .stunt file. Stuntman execution data and analytical results are added to that same .stunt file, maintaining continuity between what was requested, what ran, and what was measured in a single, structured file.

How Stuntman AI supports workflow development

Starting from natural-language descriptions of the experimental objective and workflow, Stuntman AI helps create and refine the Python scripts for configuration handling, design generation, execution, analysis, evaluation, and optimization.

Those scripts can be reviewed and validated before the experiment begins. During the run, the validated scripts coordinate the loop and the optimization engine selects the conditions.

Inside one cycle, step by step

closed_loop.py reads the configuration and passes the search space and objective to optimizer.py. The same sequence repeats until the stopping criterion is met:

1. Decide (optimizer.py)
Uses the search space, objective, and available results to propose the first or next candidate batch.

2. Update (update.py)
Makes the candidate conditions available to generate_design.py.

3. Write Design (generate_design.py)
Builds a Python design from the selected conditions and converts it into an executable .stunt file.

4. Run Experiment
The Stuntman API initiates the design, and Stuntman executes it. Stuntman prepares the experiments, processes samples, removes analytical samples for analysis and transfers them to integrated analytical equipment.

5. Analyze Data (analysis.py)
Transforms each analytical result into parameter values used by the objective function.

6. Evaluate (evaluate.py)
Checks the results against the stopping criterion. If the loop continues, the results return to optimizer.py to inform the next candidate batch. If the stopping criterion is met, the loop ends.

How the optimizer decides what's next

In closed-loop experimentation, a decision engine determines what to test next. In the demonstrated workflow, a Gaussian-process Bayesian optimizer used the conditions already tested and their objective scores to build a surrogate model. For untested conditions, the model estimated both expected performance and uncertainty.

Candidate selection balanced exploitation, refining regions that already perform well, with exploration of less-certain regions that may contain a better solution. After each cycle, the new measurements updated the surrogate model before the next candidates were proposed.

The closed-loop architecture is not limited to Bayesian optimization. Other decision engines can be substituted when a workflow calls for design of experiments, rule-based logic, or a custom model.

How the reaction optimizer was configured

For the sucrose hydrolysis workflow, the Gaussian- process optimizer was implemented with scikit-optimize. It was configured for 12 candidates per cycle for up to 12 cycles: one fixed control, the best-performing non-control condition identified so far, and 10 newly proposed conditions.

The exploration rate was scheduled at 90% for cycles 1–4, 50% for cycles 5–8, and 10% for cycles 9–12 (Figure 4). The schedule deliberately shifts the search from broad coverage in earlier cycles toward refinement in later cycles.

Each candidate was scored against reaction velocity and linearity thresholds using the normalized composite score. The pass thresholds were velocity ≥ 10 µM/sec, linearity ≥ 0.99, and composite score ≥ 1.

$$Score = 0.6 \times \left(\frac{Velocity}{Pass\ Velocity}\right) + 0.4 \times \left(\frac{Linearity}{Pass\ Linearity}\right)$$

```
# Exploration schedule: (1-4) 90.0 (5-8) 50.0 (9-12) 10.0

import pickle
from typing import Optional
from scipy.stats import qmc
from skopt import Optimizer
from skopt.learning import GaussianProcessRegressor
from skopt.learning.gaussian_process.kernels import Matern
from skopt.space import Real

class OptimizationMain:

    # ------------------------------------------------------
    # Constants
    # ------------------------------------------------------

    CANDIDATES_PER_CYCLE = 12
    OPTIMIZER_CANDIDATES = 10

    CONTROL = {'Invertase': 30, 'pH': 7.5, 'NaCl': 100}

    ORIGINAL_BOUNDS = {
        'Invertase': (5.0,  100.0),
        'pH':        (6.0,    9.0),
        'NaCl':      (0.0, 2500.0),
    }
```

Figure 4: Excerpt from optimizer.py showing the finalized exploration schedule and original bounds.

From a decision to an executable run

Once a candidate batch has been selected, the loop translates optimizer output into a design Stuntman can execute. update.py makes the conditions available to generate_design.py, which creates the executable .stunt design.

Stopping and traceability

evaluate.py checks the configured stopping logic after every cycle. The demonstrated run used the maximum cycle count as its stopping criterion, although performance thresholds can also be configured. The user can also terminate the experiment before a configured stopping criterion is reached. Whether the loop continues or ends, its inputs, selected conditions, execution data, measurements, and results remain available for review in the .stunt file.

Example: How the physical experiment ran

For the sucrose hydrolysis workflow, each cycle contained 12 reactions. Four samples were collected from each reaction at 0, 10, 20, and 30 minutes, creating the absorbance time course used to calculate reaction velocity and linearity. For this demonstrated workflow, Stunner served as the connected analytical instrument, measuring UV/Vis absorbance.

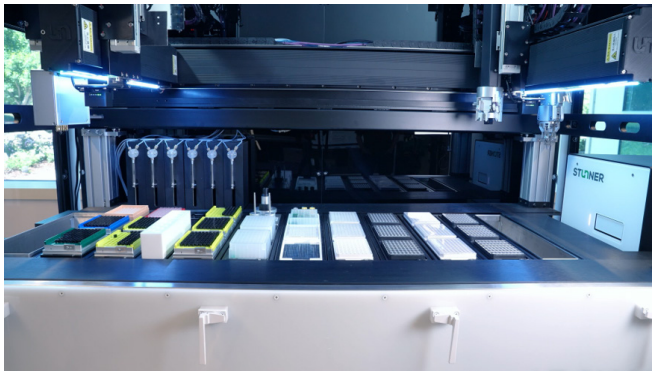

Figure 5: The Stuntman deck staged with reaction, quench, and analysis plates during the closed-loop run.

Deck layout: A reaction plate held the Reaction Mixtures library, a quench plate held the Quenched Reactions library, and a Stunner plate served as the analysis plate (Figure 5).

Automated sequence: The Stuntman API initiated the .stunt design. Stuntman dispensed and mixed the reagents, collected each timepoint, transferred it to the quench plate to stop the reaction, and moved the quenched samples to the Stunner plate for analysis.

Stunner measured UV/Vis absorbance, and analysis.py converted each absorbance time course into the reaction velocity and linearity values returned to the optimizer.

Adapting the closed-loop

The closed-loop architecture can be transferred to a new workflow.

What can stay fixed: The main-loop pattern, state management, validated handoffs, and the requirement to return results to the optimizer.

What changes by workflow: Materials and variable ranges, deck configuration, execution method, integrated analytics, analytical readout, objective function, optimizer, and stopping logic.

Where it applies: The modularity built into the closed-loop architecture supports reaction screening, optimization, and preformulation in small molecules & chemistry; electrolyte formulation, polymer development, and catalyst screening in materials & energy; and multi-parameter developability studies in biologics & gene therapy.

Conclusion

Stuntman supplies the modular automation hardware and software architecture needed to turn each new decision into a physical experiment. Configuration files define the experiment, validated Python scripts coordinate the handoffs, the selected decision engine chooses conditions, and the Stuntman API initiates each executable .stunt design for execution on Stuntman. Together, these elements create a traceable closed-loop system that can be adapted to different experiments, analytical readouts, and decision engines.

Unchained Labs
4747 Willow Road
Pleasanton, CA 94588
Phone: 1.925.587.9800
Toll-free: 1.800.815.6384
Email: info@unchainedlabs.com

Rev A