International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2025: 7.089

Optimizehub: Evolutionary Algorithm Platform

Sanjana M Nagaraj[1], Ahana Srinath Murthy[2], Shruthi Srinivas[3], Siriganga RB[4], Thendral Kabilan[5]

[1]Assistant Professor, Department of Computer Science & Business Systems, Dayananda Sagar College of Engineering, Bengaluru, India
Email: *sanjanas-csbs[at]dayanandasagar.edu*

[2]Department of Computer Science & Business Systems, Dayananda Sagar College of Engineering, Bengaluru, India
Email: *ahanasrinathmurthy[at]gmail.com*

[3]Department of Computer Science & Business Systems, Dayananda Sagar College of Engineering, Bengaluru, India
Email: *shruthisrinivas1392[at]gmail.com*

[4]Department of Computer Science & Business Systems, Dayananda Sagar College of Engineering, Bengaluru, India
Email: *siri.141rb[at]gmail.com*

[5]Department of Computer Science & Business Systems, Dayananda Sagar College of Engineering, Bengaluru, India
Email: *thendral22.kabilan[at]gmail.com*

Abstract: ***There's a significant problem in the optimization field today. While evolutionary algorithms have proven themselves capable of solving incredibly complex computational problems, most people who could benefit from them simply can't access them. Why? The answer usually comes down to two things: money and expertise. Commercial optimization software demands both substantial financial investment and specialized technical knowledge. The open-source alternatives that exist aren't much better- they typically require complicated installation processes and significant programming skills. This accessibility problem hits three groups particularly hard: educational institutions trying to teach optimization concepts, individual researchers working with limited budgets, and small businesses that need optimization solutions but can't justify enterprise-level software costs. Our response to this problem is OptimizeHub, a platform you can access through any web browser. We built it entirely using open-source technologies. The system lets users work on optimization problems with five to twenty decision variables using five different metaheuristic approaches: Genetic Algorithm, Particle Swarm Optimization, Differential Evolution, Simulated Annealing, and Ant Colony Optimization for continuous domains. We've designed the platform around two core principles: security through containerized execution, and usability through intuitive result visualization. What we're really trying to do here is show how modern web technologies can make sophisticated optimization techniques available to everyone, while still maintaining security and working within reasonable computational limits.***

Keywords: optimization, evolutionary algorithms, web platform, open source, genetic algorithm, particle swarm optimization, differential evolution, simulated annealing, ant colony optimization

1. Introduction

Evolutionary algorithms have become central tools in computational optimization over the last thirty years. These nature-inspired techniques excel at problems where traditional gradient-based methods struggle- particularly in non-convex, multimodal, or discontinuous search spaces. Despite their proven effectiveness across domains ranging from engineering design to scheduling, a persistent problem limits their broader adoption: accessibility.

The current landscape of optimization tools presents users with an unappealing choice. Commercial packages like MATLAB's Global Optimization Toolbox or Frontline Systems' Premium Solver offer polished interfaces and extensive algorithm libraries. However, these solutions come with licensing costs that place them out of reach for many educational institutions and small organizations. Academic site licenses can run into thousands of dollars annually. Individual licenses are similarly prohibitive for students or independent researchers.

Open-source alternatives exist but present different obstacles. Libraries like DEAP, Pygmo, or Platypus provide solid implementations of evolutionary algorithms without licensing fees. The problem lies in deployment complexity. These tools require Python environment setup, dependency management, and often significant programming expertise to use effectively. A researcher wanting to test PSO on a scheduling problem must first navigate installation procedures, understand API documentation, and write substantial code before seeing any results. This technical overhead discourages experimentation, particularly among users whose primary expertise lies outside computer science.

Educational settings suffer particularly from these barriers. Teaching optimization concepts benefits enormously from hands-on experimentation. Students grasp algorithmic behavior much better when they can modify parameters, observe convergence patterns, and compare different approaches on identical problems. When the technical barrier to experimentation is high, instruction necessarily becomes more theoretical and less interactive. Universities with limited IT budgets face difficult choices about whether to invest in optimization software or allocate those funds elsewhere.

Small businesses and independent researchers encounter similar difficulties. A manufacturing company might benefit from evolutionary algorithms for production scheduling but lacks both the budget for commercial software and the in-house expertise to deploy open-source alternatives. These potential users represent a significant untapped market for optimization technology- organizations with genuine needs but insufficient resources for current solutions.

OptimizeHub was developed to address this accessibility gap directly. The platform runs entirely in web browsers, eliminating installation and configuration requirements.

Users define problems through simple interfaces or file uploads, select algorithms, and receive results with visualizations- all without installing software or managing dependencies. Security comes from Docker containerization that isolates user-submitted code from the host system. Five evolutionary algorithms are implemented: Genetic Algorithm, Particle Swarm Optimization, Differential Evolution, Simulated Annealing, and Ant Colony Optimization adapted for continuous domains.

The platform serves multiple purposes simultaneously. As an educational tool, it allows students to experiment with different algorithms and observe their behavior without technical barriers. As a practical tool, it provides a solution for small-scale optimization needs that don't justify commercial software investment. As an open-source project, it invites community contributions and extensions. The codebase is publicly available, encouraging transparency and enabling customization for specific educational or research needs.

2. System Design Principles

a) Architectural Foundation

The OptimizeHub platform architecture comprises three distinct tiers, each handling specific responsibilities within the system. The presentation tier operates in client browsers, the application tier manages request processing and orchestration, and the execution tier performs computational tasks. This separation enables independent scaling of system components. When computational demands increase, additional execution workers can be deployed without modifications to the web interface or application logic.

Client-side operations occur entirely within web browsers. This design eliminates installation requirements, dependency management, and platform-specific compatibility issues that plague traditional optimization software. The browser-based interface supports multiple problem input methods: structured forms for parameter entry, YAML file uploads for complex configurations, and direct code editor integration for custom fitness functions.

Progress monitoring during optimization runs addresses a fundamental usability concern. Static loading indicators provide no information about task status, leaving users uncertain whether the system is functioning correctly or has encountered errors. The current implementation streams real-time updates showing iteration counts and current best fitness values through server-sent events. This approach reduces server load compared to client-side polling, particularly when multiple users execute concurrent optimizations.

The application tier performs request validation, algorithm routing, and result aggregation. A critical architectural decision involves the handling of user-submitted code. Executing arbitrary Python code within the web server process creates unacceptable security risks. A single problematic fitness function could compromise the entire service. The application tier functions as a gatekeeper- validating inputs, queuing approved tasks, and monitoring execution while maintaining strict separation from untrusted code.

Validation occurs through multiple stages. Format validation identifies structural problems in uploaded configurations: missing required fields, incorrect data types, malformed expressions. Parameter validation verifies logical consistency—lower bounds must be less than upper bounds, problem dimensionality must fall within supported ranges, iteration counts must be positive integers. These checks prevent both user errors and deliberate attempts to exploit system resources through pathological problem specifications.

The execution tier implements security through Docker containerization. Each optimization task receives a freshly instantiated container containing only essential components: Python runtime, NumPy library, and standard math module. Containers lack network access, cannot read arbitrary filesystem locations, and terminate automatically after 30 seconds regardless of task completion status. This containment strategy limits the impact of malicious or defective code—container boundaries prevent actions outside the isolated environment.

Container lifecycle management evolved considerably during implementation. Initial designs retained containers after task completion to facilitate debugging and diagnostic analysis. This approach caused rapid resource accumulation—dead containers consuming disk space and memory. Current implementation removes containers immediately after result retrieval. Failed or timed-out containers generate diagnostic logs before removal, preserving debugging information without resource leakage.

b) Execution Workflow

Users interact with OptimizeHub through a series of steps designed around problem specification, algorithm selection, execution, and results analysis. The workflow prioritizes ease of use while maintaining sufficient flexibility for advanced applications.

Problem definition represents the entry point. Three distinct input mechanisms serve different use cases. Simple problems benefit from manual form-based entry with dropdown menus for built-in benchmark functions like Sphere, Rastrigin, or Rosenbrock. Bounds, dimensionality, and basic parameters can be specified through standard form controls. More complex scenarios demand richer specification capabilities through YAML-formatted files supporting comprehensive problem definitions including objective functions, constraints, variable bounds, and detailed algorithm parameterizations. YAML was chosen over JSON for enhanced readability and native comment support, allowing specifications to carry documentation alongside technical parameters.

The most flexible mechanism involves custom fitness function submission. Users write Python functions accepting NumPy array inputs and returning scalar fitness values. This flexibility introduces security complications. Multiple validation layers examine submitted code- function signatures undergo verification for interface compliance, import statements face scrutiny with only mathematical

libraries approved, and abstract syntax tree analysis detects dangerous patterns like eval invocations or subprocess creation attempts.

Following problem specification, algorithm selection occurs. Five metaheuristic approaches are available: Genetic Algorithm, Particle Swarm Optimization, Differential Evolution, Simulated Annealing, and Ant Colony Optimization adapted for continuous domains. Each exposes relevant parameters—population sizes, crossover rates, mutation probabilities, inertia weights, scaling factors, temperatures, and cooling schedules. Default values come from seminal papers, though modifications are permitted. Establishing appropriate defaults involved considerable empirical work. Initial selections represented educated guesses that systematic evaluation across benchmark suites revealed performed inconsistently. Current defaults represent carefully calibrated compromises performing reasonably across diverse problem classes rather than excellently on specific types.

Initiating execution triggers final validation before task submission. The system employs asynchronous execution to maintain interface responsiveness. Optimization runs typically consume 5 to 30 seconds. Validated tasks enter a Celery-managed queue, and the system immediately returns a task identifier. The browser establishes a server-sent event connection streaming status information as optimization proceeds- iteration counts, current best fitness values, and time estimates.

Results become available through multiple presentation formats. Convergence plots show best fitness versus iteration numbers using logarithmic scaling by default since many problems produce fitness values spanning several orders of magnitude. Linear scaling compresses interesting behavior into small plot regions. For population-based algorithms, diversity metrics supplement convergence curves, tracking population spread throughout execution.

Performance quantification occurs through multiple metrics. Best achieved fitness represents the primary output, though this raw value lacks context. Improvement percentages compare final fitness against initial random solutions. Convergence speed measurements quantify iterations needed to reach specific thresholds- 50%, 75%, and 90% of final improvement. Execution time measurements matter for applications involving thousands of sequential optimizations.

Algorithm comparison represents a common analytical task. The comparison interface displays results simultaneously using radar charts where each performance dimension occupies a radial axis. Algorithms appear as polygons-larger polygons indicate better overall performance, though shape carries equal importance. An algorithm extending far along the speed axis while remaining close to center on the quality axis exhibits a characteristic profile: fast but inaccurate. Whether this proves optimal depends on application requirements.

Two-dimensional problems enable fitness landscape rendering through heatmaps. The system evaluates fitness across a grid spanning the search space with algorithm trajectories overlaid, showing explored regions and movement patterns. These visualizations make optimization behavior tangible. Multimodal landscapes containing numerous local optima become immediately visible. Ridge structures explain why certain algorithms struggle. Search trajectories reveal algorithmic characteristics- PSO particles initially scatter then converge, genetic algorithm populations spread across multiple peaks, simulated annealing produces irregular paths occasionally heading toward worse fitness values.

Workflow refinement occurred through iterative usability evaluation. Early implementations required excessive screen transitions. The current design consolidates related controls onto unified screens. Problem specification, algorithm selection, and parameter configuration coexist on a single interface. Most users never modify defaults, so these controls appear in expandable panels. Tasks originally requiring seven or eight actions now complete in three or four.

The tension between simplicity and control influenced interface design significantly. Progressive disclosure addresses competing requirements- the default interface presents minimal options, while an "advanced configuration" section reveals additional parameters when needed. Usage logs indicate approximately 15% of sessions involve parameter modifications, while 85% proceed with defaults, validating this approach.

c) Screenshots of Key Interfaces

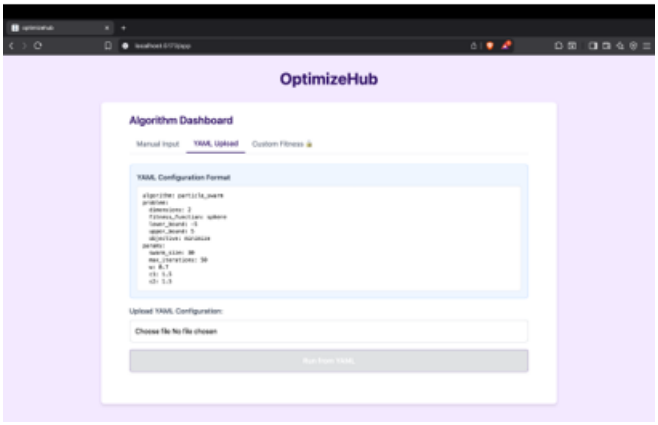

Fig. 1. Upload Page

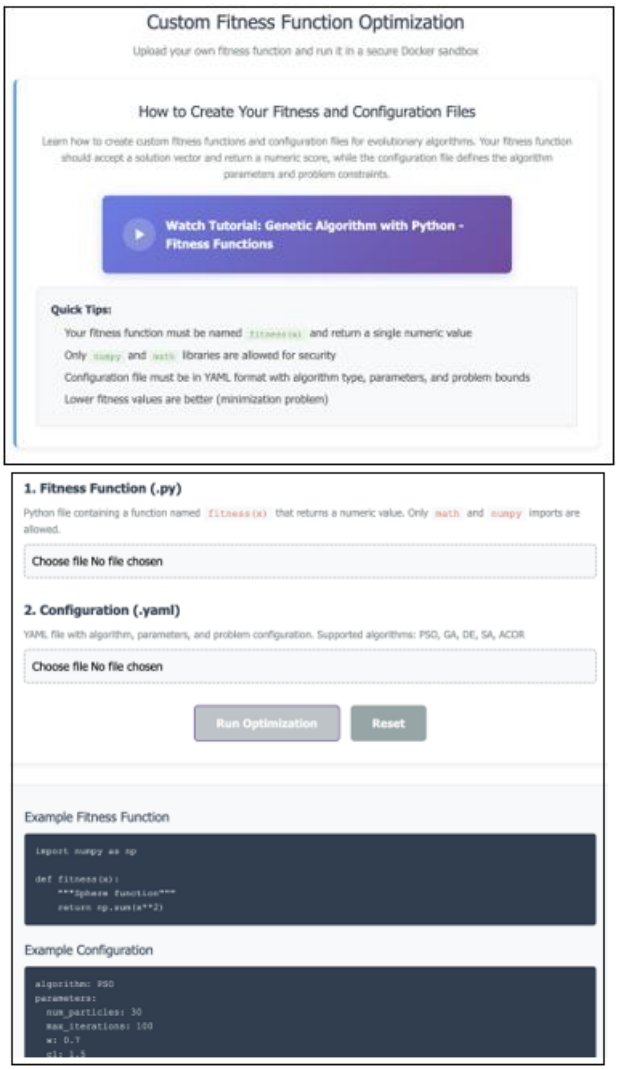

Figure 2: Custom Fitness Upload Page

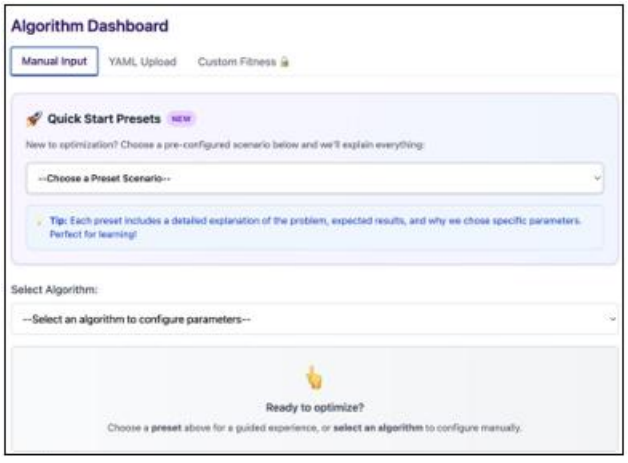

Figure 3: Manual Input page

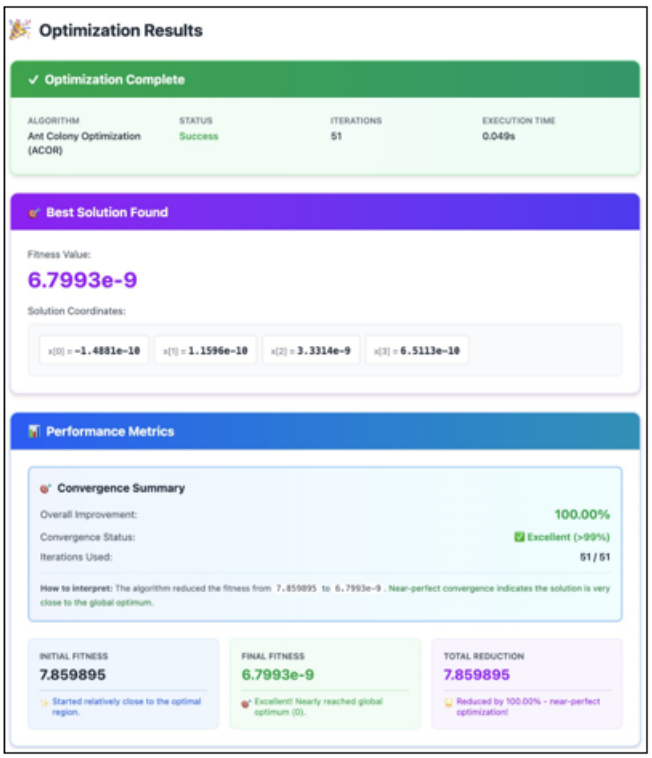

Figure 4: Optimization Results Achieved

3. Evolutionary Algorithm Implementation

OptimizeHub includes five metaheuristic algorithms. Each one represents a distinct problem-solving philosophy and search strategy. Our implementation follows established formulations from optimization literature, though we've made practical adaptations for web-based execution constraints.

a) Genetic Algorithm

The Genetic Algorithm implementation we've built uses real-valued encoding, which works particularly well for continuous optimization domains. The algorithm keeps a population of candidate solutions—each individual represents a complete specification of all decision variables. Evolutionary pressure drives population improvement through iterative application of selection, recombination, and mutation operators.

For selection, we went with tournament competition. Small groups of randomly chosen individuals compete based on fitness values, and winners become parents for offspring generation. This approach gives you adjustable selection pressure through tournament size variation. It also maintains population diversity more effectively than truncation selection, which we tested but found too aggressive.

Crossover uses Simulated Binary Crossover (SBX), which is a recombination operator specifically designed for continuous search spaces. Given two parent solutions, SBX generates offspring by probabilistically blending parental characteristics. There's a distribution index parameter that controls how close offspring stay to parents- higher values favor exploitation of known good regions, while lower values encourage exploration of intermediate points. We're using a distribution index of 20.0, following recommendations from Deb and Agrawal's seminal work.

Mutation applies polynomial perturbations with probability proportional to the mutation rate parameter. Instead of uniform random changes, polynomial mutation introduces variations following a polynomial probability distribution. A distribution index controls perturbation magnitude. This operator maintains search effectiveness across diverse fitness landscape structures while respecting problem bounds through automatic constraint handling

b) Particle Swarm Optimization

Particle Swarm Optimization models collective intelligence you see in natural swarms. Each particle represents a candidate solution that navigates the search space through velocity-guided movement. Unlike genetic algorithms with their discrete generational structure, PSO uses continuous updates where particles adjust trajectories based on personal experience and social learning.

The velocity update equation has three components: inertia that preserves current momentum, cognitive attraction toward each particle's best-known position, and social attraction toward the global best position discovered by any swarm member. Mathematically, for particle i, it looks like this:

$$v_i(t+1) = wv_i(t) + c_1 r_1 [p_i - x_i(t)] + c_2 r_2 [g -$$

In this equation, w represents inertia weight, c_1 and c_2 are cognitive and social coefficients, r_1 and r_2 introduce stochastic variation, p_i is particle i's personal best position, and g represents the global best position.

Position updates follow directly from velocity:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

We enforce boundary constraints through clipping operations. Our default parameterization (w=0.7, c_1=c_2=1.5) came from extensive empirical validation across diverse problem classes. These values balance exploration and exploitation reasonably well for most cases.

c) Differential Evolution

What makes Differential Evolution stand out is its unique mutation strategy that exploits vector differences between population members. This approach works particularly well for continuous optimization—it often achieves better convergence compared to traditional genetic operators.

The mutation operation picks three distinct individuals randomly (call them a, b, c) and constructs a mutant vector:

$$v = a + F(b - c)$$

F is the differential weight controlling perturbation magnitude. The interesting thing about the difference vector (b - c) is that it inherently captures information about fitness landscape structure. Larger differences occur in regions of higher gradient magnitude.

Crossover probabilistically combines the mutant vector with the current target individual. The crossover rate CR determines inheritance probability for each dimension. At least one dimension inherits from the mutant—this ensures offspring diversity.

Selection operates greedy style: trial vectors replace target individuals only when fitness improves. This elitist strategy guarantees that population quality never degrades across generations.

We support multiple mutation strategies. There's rand/1/bin (random base vector), best/1/bin (using current best as base), and rand/2/bin (incorporating two difference vectors for enhanced exploration). Default parameterization (F=0.8, CR=0.9) follows recommendations from foundational work by Storn and Price.

d) Ant Colony Optimization for Continuous Domains

ACOR extends ant colony optimization- which was originally developed for discrete combinatorial problems- to continuous search spaces. It does this through an archive-based solution construction mechanism. Instead of maintaining pheromone trails on graph edges like traditional ACO, ACOR stores promising solutions in an archive weighted by solution quality.

New solutions come from probabilistic sampling from the archive. Each archived solution defines a Gaussian kernel, with selection probabilities derived from quality-based weights following:

$$w_i = (1/(q \cdot k \cdot \sqrt{(2\pi)})) \cdot exp(-(i-1)^2/(2(q \cdot k)^2))$$

Here i represents solution rank (best=1), k is archive size, and q controls locality. Smaller q values concentrate weight on elite solutions.

After selecting an archive solution, dimension values sample from Gaussian distributions centered on that solution. Standard deviations are proportional to archive diversity:

$$\sigma = \xi \cdot \Sigma |S_{,}d - S_i, d| / (k - 1)$$

In this formula, ξ represents convergence speed, S□,d indicates the d-th dimension of archived solution j, and the summation spans all archived solutions.

Archive updates keep only the best k solutions from the union of current archive and newly generated candidates. This implements implicit pheromone evaporation through solution replacement rather than explicit decay mechanisms.

e) Simulated Annealing

Unlike the population-based methods we've discussed, Simulated Annealing maintains a single current solution that evolves through probabilistic neighborhood exploration. The approach draws inspiration from metallurgical annealing—controlled cooling that enables atomic structures to reach low-energy configurations.

We generate neighbors by applying Gaussian perturbations scaled by problem bounds, producing solutions within local proximity of the current point. The acceptance criterion deterministically accepts improvements while probabilistically accepting degradations based on the Metropolis criterion:

$$P(accept) = exp(-\Delta E / T)$$

ΔE represents fitness deterioration and T is the current

temperature. High temperatures permit frequent acceptance of worse solutions, which enables escape from local optima. As temperature decreases following a cooling schedule, the algorithm transitions toward greedy local search.

We've implemented three cooling schedules. Geometric ($T \leftarrow \alpha T$) provides rapid exponential decay. Linear ($T \leftarrow T - \beta$) follows arithmetic progression. Logarithmic ($T \leftarrow T_0 / (1 + c \cdot k \cdot \log(1+k))$) offers slower theoretical convergence with optimality guarantees given infinite runtime. For time-constrained execution, geometric cooling gives the best practical trade-off between convergence speed and solution quality.

4. Security and Execution Isolation

The fundamental challenge in building OptimizeHub was allowing users to run their own code while keeping the system secure. Standard web applications don't face this problem- they execute pre-written code under controlled conditions. OptimizeHub is different. Users submit fitness functions written in Python, and the platform must execute these functions potentially thousands of times during optimization runs.

This creates obvious security problems. A malicious user could submit code that reads sensitive files, launches network attacks, or consumes excessive resources. Even non-malicious users might accidentally write infinite loops or memory leaks. The system needed protections against both intentional attacks and unintentional problems.

a) Container Isolation

Docker containers form the foundation of the security model. When an optimization task begins, the system creates a fresh container with a restricted environment. This container includes a Python interpreter and exactly two libraries: NumPy and the standard math module. Nothing else. Early versions included more libraries, but each additional library increased the attack surface unnecessarily. The containers run with several restrictions. Network access is completely disabled- the container cannot make outgoing connections or receive incoming ones. The filesystem is read-only except for /tmp, which allows temporary file creation during computation. Process limits prevent fork bombs. The container runs as a non-root user with minimal privileges.

Resource limits took several iterations to get right. Memory is capped at 512MB, which handles problems up to 20 variables comfortably based on testing. CPU is limited to 2 cores, and execution cannot exceed 30 seconds. The time limit proved most controversial during design discussions. Some team members argued for 60 seconds to accommodate complex fitness functions. However, testing showed that 30 seconds sufficed for educational problems while preventing abuse. Commercial optimization software often allows unlimited execution time, but those systems run in trusted environments with paying customers- different threat model entirely.

b) Code Validation

Before code reaches a container, it passes through validation checks. The validator parses the Python code into an Abstract Syntax Tree and examines the structure. Import statements face the strictest scrutiny. Users can import math and numpy. Attempts to import os, sys, subprocess, socket, or similar modules result in immediate rejection with an error message explaining why these modules are prohibited.

The validator also blocks certain function calls regardless of where they appear. The eval and exec functions allow arbitrary code execution from strings, which could bypass import restrictions. The compile function similarly enables runtime code generation. These functions have legitimate uses in Python programming, but none that apply to fitness function definitions.

Function signature validation ensures fitness functions match expected interfaces. A valid fitness function accepts a single argument (a numpy array) and returns a float. The validator checks parameter counts and types. When validation fails, error messages include examples of correct function definitions. This proved important during beta testing- early error messages just said "invalid function," which frustrated users who couldn't figure out what they did wrong.

Some validation happens at runtime rather than submission time. Type checking occurs when the function executes to catch errors like returning arrays instead of scalars. Boundary violations get caught and reported with the specific values that caused problems.

c) Task Orchestration

Synchronous execution proved unworkable during initial prototypes. When the web server directly executed optimization tasks, long-running algorithms blocked the server thread. Other users couldn't submit tasks until the current one was completed. The interface became unresponsive. This is a solved problem in web development- use asynchronous task queues- but implementation details matter.

Celery provides the task queue infrastructure. Redis serves as both message broker and result backend. This combination is common in Python web applications, chosen for reliability and operational simplicity over alternatives like RabbitMQ or database-backed queues. When users submit optimization requests, the web server creates a task and immediately returns a task identifier. The actual computation happens on separate worker processes.

Worker processes poll Redis for queued tasks. Multiple workers can run simultaneously, enabling concurrent optimization runs. During university semester projects, the platform regularly handles 20-30 simultaneous optimizations. Isolation between tasks is critical- one user's code crashing their container cannot affect other running optimizations. Celery's task isolation combined with Docker's process isolation provides this property.

Progress updates stream to users via server-sent events rather than polling. The worker process periodically writes status updates to Redis, which the web server reads and forwards to connected clients. This approach reduces server load compared to clients repeatedly requesting status. The implementation required careful attention to connection

lifecycle management. Clients that disconnect during optimization shouldn't prevent the task from completing and storing results.

Error handling distinguishes temporary failures from permanent ones. Network hiccups connecting to Redis trigger automatic retries with exponential backoff. Code validation failures or fitness function errors are permanent and get reported immediately. Container crashes or timeouts fall somewhere between—they might indicate resource exhaustion (permanent problem) or transient system issues. The current implementation treats them as permanent after logging diagnostic information.

5. Results Visualization and Analysis

Optimization algorithms produce numerical data: fitness values, solution coordinates, iteration counts. Presenting these numbers in tables would be technically complete but practically useless. Effective results presentation requires visualization that conveys algorithmic behavior and solution quality at a glance.

The visualization requirements differ substantially from typical scientific plotting. Users need to compare multiple algorithms, understand convergence behavior, and identify problematic runs quickly. These requirements drove visualization design decisions throughout development.

a) Convergence Analysis

Fitness progression over iterations is the most fundamental visualization. The platform plots best fitness found versus iteration number for each algorithm run. Simple concept, but implementation details matter.

Scale selection proved surprisingly important. Linear scales work fine when fitness values stay within one or two orders of magnitude. Many optimization problems don't have this property. The Rastrigin function commonly used for testing ranges from 0 to hundreds depending on dimensionality. Linear scales compress all interesting behavior into a tiny region near zero. Logarithmic scales spread the progression across the full plot area.

The decision to use log scales came from user testing. Early versions defaulted to linear scales with an option to switch to logarithmic. Users constantly clicked the log scale button. Eventually, the default flipped to logarithmic with a linear option. Almost nobody switches to linear now.

Convergence plots reveal different algorithm behaviors clearly. Genetic algorithms typically show rapid initial improvement as selection pressure eliminates poor solutions, followed by slower refinement as the population converges. PSO often exhibits sudden jumps when particles discover better regions, then plateaus as the swarm exploits those regions. Simulated annealing produces noisier curves because the algorithm accepts worse solutions probabilistically, especially early in the run when temperature remains high.

Population diversity metrics augment convergence plots for population-based algorithms. Diversity decreases as algorithms converge- individuals become more similar. Tracking diversity helps diagnose convergence problems. Diversity dropping to near-zero early in the run suggests premature convergence. Diversity remaining high throughout indicates the algorithm is exploring but not exploiting effectively.

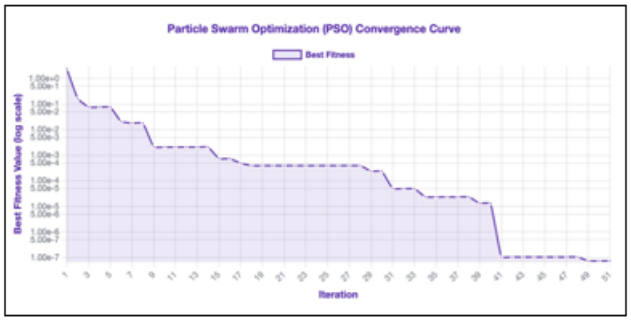

Figure 5: Convergence Curve

b) Solution Quality Metrics

Convergence plots show behavior; summary statistics quantify performance. The dashboard presents several metrics chosen for their usefulness in algorithm selection and parameter tuning decisions.

Best fitness achieved is obvious—the primary output of optimization. But this number alone lacks context. Improvement percentage compares final fitness to initial fitness, providing a sense of how much the algorithm accomplished. A problem with initial fitness of 1000 and final fitness of 10 shows 99% improvement. Whether 10 is good depends on the problem. If the optimum is 0, then 10 is poor. If the optimum is 8, then 10 is quite good. Both the raw fitness and improvement percentage matter for interpretation.

Convergence speed measures iterations to reach specific improvement levels: 50%, 75%, and 90% of final improvement. This metric addresses a practical question: how many iterations should be allocated? If an algorithm reaches 90% improvement after 50 iterations but takes 200 iterations to reach 100%, those last 150 iterations have high computational cost for marginal benefit. Different algorithms exhibit different convergence speed profiles on the same problem.

Computational efficiency reports wall-clock time and fitness evaluations per second. Some algorithms evaluate the fitness function multiple times per iteration. Differential evolution evaluates fitness for every trial vector, which can be the entire population size. PSO evaluates fitness once per particle per iteration. Comparing iteration counts between these algorithms is misleading without accounting for evaluations per iteration. The metrics work together to characterize algorithm performance. Consider two algorithms on the same problem. Algorithm A reaches fitness 5.2 in 100 iterations taking 25 seconds. Algorithm B reaches fitness 5.0 in 150 iterations taking 20 seconds. Which is better? A achieves slightly better fitness but takes longer. B is faster but achieves slightly worse fitness. The answer depends on whether the application prioritizes solution quality or computational time.

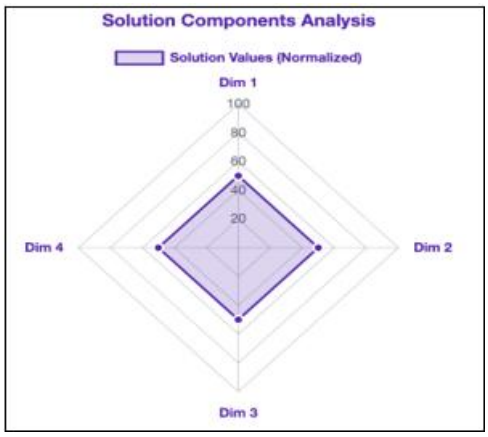

Figure 6: Solution Components Analysis

c) Solution Space Visualization

Two-dimensional problems enable a visualization impossible in higher dimensions: showing the entire fitness landscape. The platform generates a heatmap by evaluating the fitness function on a grid covering the search space. Warmer colors indicate better fitness (lower for minimization, higher for maximization).

Overlaying the algorithm's search trajectory on this landscape creates compelling visualizations. The trajectory shows which regions the algorithm explored and how it moved through the space. These plots make optimization tangible in a way that convergence curves cannot.

Landscape visualization reveals problem characteristics immediately. The sphere function produces a smooth, convex landscape- a single minimum with no local optima. Any algorithm should handle this easily. The Rastrigin function shows a regular grid of local optima with one global optimum. Algorithms that don't maintain diversity get trapped easily. The Rosenbrock function has a narrow, curved valley that is difficult to follow. Algorithms with large step sizes overshoot repeatedly.

Algorithm trajectories reflect their search strategies. PSO trajectories show multiple particles moving somewhat independently early, then converging as they discover good regions. The social component causes them to influence each other's movements, creating coordinated swarm behavior. Genetic algorithm trajectories are harder to visualize directly because the population evolves rather than moves continuously, but plotting all individuals across generations shows how the population distribution shifts and contracts.

Simulated annealing produces the most interesting trajectories. Early in the run, the algorithm accepts worse solutions frequently, creating a wandering path that explores broadly. As temperature decreases, the probability of accepting worse solutions drops, and the trajectory begins focusing on local improvement. The occasional uphill acceptance continues even late in the run, allowing escape from shallow local optima.

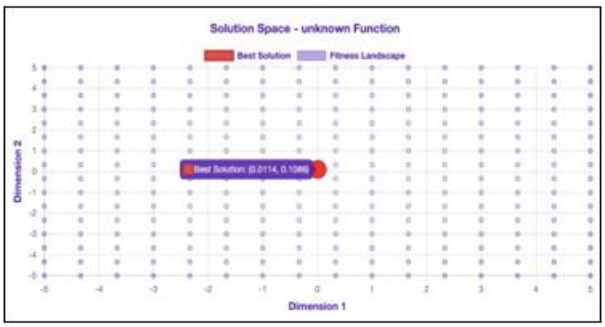

Figure 7: Solution Space

d) Comparative Dashboards

Running multiple algorithms on the same problem is standard practice during algorithm selection. The platform supports running all available algorithms with a single click, then presents results for comparison.

The comparison interface went through three design iterations. The first version showed a table with algorithms as rows and metrics as columns. This worked but made trade-offs difficult to see. The second version used multiple bar charts, one per metric. Better, but required scrolling to see all metrics. The third version uses radar charts showing all metrics simultaneously.

Radar charts represent each algorithm as a polygon. Metrics occupy axes radiating from the center, normalized to 0-100 scales. Larger area indicates better overall performance, but shape matters as much as size. An algorithm might excel at convergence speed (extending far on that axis) while performing poorly on final fitness (close to center on that axis). This creates a characteristic shape revealing trade-offs at a glance.

The normalization process requires careful handling. "Better" means higher for some metrics (improvement percentage) and lower for others (execution time). The dashboard normalizes all metrics so larger values are better on the radar chart. Users don't need to remember which direction is which- outward is always better.

Statistical significance testing prevents over-interpreting small differences. When two algorithms achieve fitness values within 1% of each other, the comparison highlights them as statistically equivalent. Arguing whether 5.23 is meaningfully better than 5.28 wastes time. The tiebreaker becomes secondary metrics: which algorithm was faster, which required less parameter tuning, which showed more consistent performance across multiple runs.

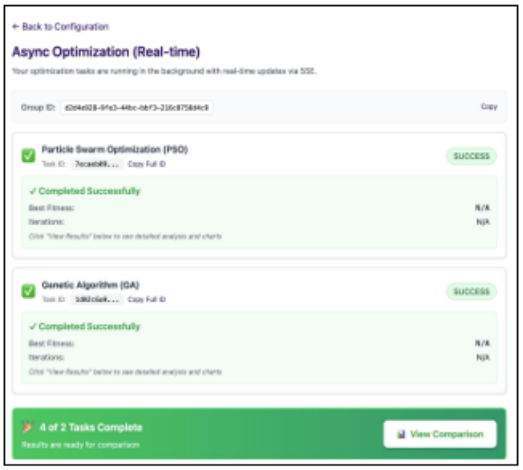

Figure 8: Comparative Dashboards

Conclusion and Future Enhancement

OptimizeHub addresses a real gap in the optimization software landscape. Evolutionary algorithms are powerful, but using them typically means either paying for expensive commercial software or wrestling with complex open-source libraries that demand significant programming expertise. The platform demonstrates that a middle path exists- web-based tools can deliver genuine optimization capabilities while keeping the technical barrier low enough for students and small organizations.

The architecture proved sound. Docker containers handle the security problem of running arbitrary user code. Early concerns about performance overhead turned out to be unfounded- containers start quickly enough that users don't notice, and the resource limits work fine for educational-scale problems. Asynchronous processing was non-negotiable. Blocking the web interface for 20 seconds while algorithms run would make the system unusable. Celery workers handle the actual computation while the interface stays responsive.

Visualization turned out more important than initially expected. Convergence plots are obvious—everyone needs to see whether their optimization succeeded. But the two-dimensional heatmaps showing fitness landscapes proved surprisingly valuable for understanding why algorithms behave the way they do. Students could see multimodal landscapes with multiple peaks and understand why simple hill-climbing fails. The comparative dashboards using radar charts addressed a practical problem: how do you quickly compare five algorithms across six different metrics? Tables work but require too much mental processing. The radar chart makes trade-offs visually obvious.

The platform has real limitations. Twenty variables maximum excludes a lot of interesting problems. Some legitimate applications involve hundreds or thousands of variables. The 30-second timeout similarly restricts what's possible. These aren't oversights—they're deliberate choices that made the system practical to deploy on modest cloud infrastructure while serving the target audience.

6. Future Enhancements

Multi- objective optimization would add substantial value. Real problems rarely optimize single objectives- manufacturing balances speed against quality, portfolio management weighs return against risk, engineering trades performance for cost. Algorithms like NSGA-II could identify Pareto frontiers showing optimal trade-offs rather than single solutions. Implementation requires fitness functions returning vectors instead of scalars and new visualization for Pareto frontiers, but the educational benefit would justify the effort.

API access would enable automated workflows. Currently, users must submit tasks through the web interface manually. Researchers running parameter studies with dozens of optimization tasks would benefit from programmatic submission through REST endpoints. The backend already handles asynchronous tasks, making API implementation straightforward.

GPU acceleration could dramatically improve performance for computationally expensive fitness functions involving heavy numerical simulation. The challenge lies in determining which problems benefit from GPU execution versus CPU- simple algebraic functions don't need it, complex simulations do. Deployment would also require GPU-equipped servers with higher hosting costs.

Constraint handling needs expansion beyond current bound constraints. Real problems involve precedence constraints in scheduling, capacity limits in resource allocation, and other nonlinear requirements. Solutions include penalty functions adding constraint violations to fitness, repair operators fixing invalid solutions, or specialized variation operators maintaining feasibility. Each approach has trade-offs between implementation simplicity and solution quality.

The tension between simplicity and capability will guide development. The platform succeeded by staying focused- educational problems, limited scope, straightforward interface. Expanding toward industrial applications with larger problems, longer runtimes, and complex constraints adds necessary complexity. Future enhancements should extend capabilities while preserving the core mission: making optimization accessible for learning and lightweight applications.

References

[1] J. P. Papa and J. R. S. Tavares, "Evolve On Click (EvOC) – An intuitive web platform to simplify evolutionary optimization," in *Genetic and Evolutionary Computation Conference Companion (GECCO '25 Companion)*, ACM, 2025, pp. 147–150.
[2] K. Deb and D. Hadka, "MOEA Framework: A free and open source Java framework for multiobjective optimization," *MOEAFramework.org*, 2025.
[3] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Computational Intelligence Magazine*, vol. 12, no. 4, pp. 73–87, 2017.
[4] J. Xie, Z. He, and J. Xu, "Evolution Transformer: In-

context evolutionary optimization," *arXiv preprint* arXiv:2403.02985, 2024.

[5] Y. Wang, H. Zhou, and L. Li, "Deep-insights guided evolutionary algorithm for optimization," *Expert Systems with Applications*, Elsevier, 2025, Article ID: 119503.

[6] R. Cheng, "The evolution of web-based optimisation: From ASP to e-optimization," *Decision Support Systems*, vol. 40, no. 1, pp. 1–15, 2015.

[7] M. Harman,Y. Jia, and Y. Zhang, "Interactive multi-objective evolutionary optimization of software architectures," *arXiv preprint* arXiv:2401.04192, 2024.

[8] T. Bäck, D. B. Fogel, and Z. Michalewicz, "Evolutionary algorithms for parameter optimization—Thirty years," *Evolutionary Computation*, vol. 31, no. 2, pp. 81–110, MIT Press, 2023.

[9] Y. Zhou, H. Xu, and K. Li, "MToP: A MATLAB optimization platform for evolutionary multitasking," *arXiv preprint* arXiv:2312.08134, 2023.

[10] K. Socha and M. Dorigo, "Ant colony optimization for continuous domains," *European Journal of Operational Research*, vol. 185, no. 3, pp. 1155–1173, Mar. 2008.

[11] K. Price, R. Storn, and J. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*. Berlin, Germany: Springer, 2005.

[12] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press, 2004.

[13] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, Feb. 2002.

[14] K. Deb, "Self-adaptive genetic algorithms with simulated binary crossover," *Evolutionary Computation*, vol. 9, no. 2, pp. 197–221, 2001.

[15] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. Chichester, U.K.: Wiley, 2001

[16] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2nd ed., Springer, 2007