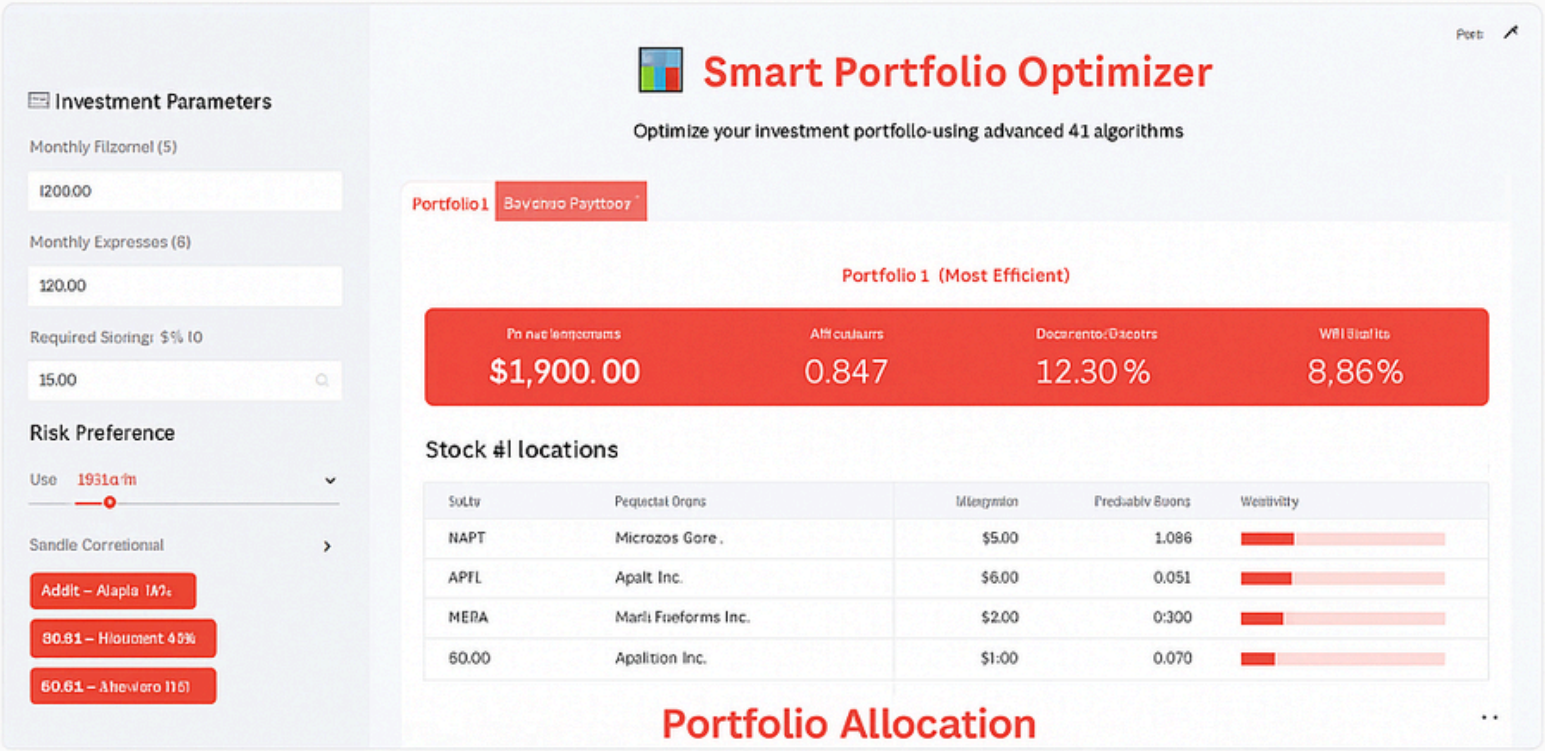
# AI-Driven Portfolio Optimization and Budget Planner





**Alina Siddiqui – 26555**
**Rafsha – 26639**
**Sheikh M Umair – 26409**
**Ibrahim Khalil – 27003**

# AI-Driven Portfolio Optimization and Budget Planner

Shaikh M. Umair, Alina Siddiqui, Rafsha Rahim, and Ibrahim Khalil Ahmad

Institute of Business Administration, Karachi, Pakistan
{26409, 26555, 26639, 27003}

# Table of Contents

# Abstract

This paper presents a novel AI-driven system for personalized portfolio optimization and budget planning. The proposed approach integrates Constraint Satisfaction Problem (CSP) solvers and heuristic-guided A* search to generate investment recommendations that honor strict budget, risk, and diversification constraints. The CSP model encodes feasibility based on user income, expenses, and minimum savings targets. In parallel, the A* search leverages predictions from stock forecasting models such as LSTM to guide the search toward high-return, executable portfolios. Together, these components ensure that the final recommendations are not only theoretically optimal but also practically aligned with the user's financial profile.



**Keywords:** Portfolio Optimization, Constraint Satisfaction, A* Search, Budget Planning, Financial AI

# 1 Introduction

Investment planning is a complex decision-making process involving uncertainty, resource limitations, and conflicting financial goals. Novice investors frequently struggle to balance income, expenses, and future savings while choosing from an overwhelming number of investment options. Static budgeting tools fail to adapt to user-specific financial situations and often ignore market trends and risk tolerances. In response, this project introduces an AI-driven portfolio planner that combines CSP-based feasibility validation with A* search and machine learning for optimal, personalized planning.

# 2 Related Work

The convergence of artificial intelligence and personal finance has produced a wide range of automated planning tools. However, most conventional systems provide either rigid budget templates or opaque recommendation engines with limited personalization. Our proposed system builds on existing literature by combining three AI paradigms—constraint satisfaction, heuristic-guided search, and predictive modeling—into a modular, explainable, and user-adaptive pipeline for portfolio optimization.

## 2.1 Constraint Satisfaction in Financial Planning

Constraint Satisfaction Problems (CSPs) offer a structured approach to enforce hard rules such as budget limits, diversification minimums, and investment ceilings. Jezeie et al. [1] proposed a knapsack-style portfolio optimization model using discrete dynamic programming. Their work incorporates cardinality constraints and budget ceilings, and outputs optimal integer solutions, which is particularly useful for dealing with expensive stocks in real-world portfolios.

While their formulation is effective for static portfolios with fixed constraints, our system extends the expressiveness of CSP by incorporating real-time variables like market-derived risk, diversification enforcement, and user-defined financial profiles (e.g., income and required savings). Moreover, we use a recursive backtracking solver instead of dynamic programming, which allows us to integrate custom constraint logic at runtime, improving modularity and extensibility.

## 2.2 A* Search in Portfolio Optimization

Gao et al. [2] applied A* search to the domain of financial planning by introducing an A* tree search model for portfolio selection. Their approach combined reinforcement learning with heuristic evaluation to select optimal paths in a dynamic environment. While this integration demonstrated strong theoretical performance, their work primarily focuses on abstract state exploration without strong ties to individual user financial constraints.

In contrast, our A* implementation operates on a space already filtered by the CSP layer, ensuring that every state considered is financially valid. Moreover, our heuristic function is explicitly tied to predictive models trained on historical data (LSTM), enabling goal-directed search tailored to user-specific stock predictions. This tight integration between feasibility and optimization enhances both relevance and efficiency.

### 2.3 Machine Learning in Financial Forecasting

Thakur et al. [3] explored the use of supervised learning models—including LSTM—for financial forecasting. They demonstrated that LSTM models are particularly effective at capturing temporal trends in stock behavior, while tree-based models offer strong short-term performance in structured data scenarios. Their work provided key insights into model selection for return prediction, but stopped short of integrating these models into an optimization pipeline.

Our system builds on their forecasting framework by directly embedding the predicted returns as heuristic drivers in the A* search algorithm. This allows machine learning outputs to actively shape the search trajectory rather than being used in isolation. Furthermore, our preprocessing, normalization, and aggregation steps ensure that sector-wise predictions are fairly integrated, overcoming the problem of model inconsistency across assets.

### 2.4 Hybrid AI Approaches in Portfolio Management

Several recent frameworks have attempted to combine different AI techniques for smarter portfolio generation. However, most hybrid systems focus on simulation-based evaluation or black-box optimization. For example, deep reinforcement learning agents have shown promise in learning investment policies, but their interpretability remains limited.

Our approach explicitly separates feasibility (CSP), optimization (A*), and forecasting (ML), enabling transparency, debuggability, and user control at each stage. This modular architecture not only simplifies tuning and testing but also allows end-users to trace how each financial constraint or predicted return influenced the final decision. By grounding each decision in both mathematical feasibility and data-driven expectation, our system offers a rare blend of explainability and adaptiveness tailored to real-world users.

## 3 Methodology

Our AI-driven financial planning system combines a Constraint Satisfaction Problem (CSP) solver and a heuristic-guided A* search engine. The CSP component ensures financial feasibility by rigorously enforcing user-defined constraints such as budget, savings goals, and risk limits, while the A* module uses stock return predictions as heuristics to optimize for return under uncertainty. This section details the CSP component.

### 3.1 Data Acquisition and Preprocessing

Stock market data was obtained from two sources. Historical price data was retrieved using the `yfinance` API, which provided adjusted daily close values for curated stock tickers. This data was used to compute financial metrics such as portfolio variance, Sharpe Ratio, and expected return.

In parallel, sector-specific heuristic scores were loaded from a structured CSV file (`Stocks - Heuristics.csv`). These scores represent precomputed estimates of expected stock performance and were curated offline for experimentation purposes. The heuristic scores in the CSV were generated offline using a Long Short-Term Memory (LSTM) model trained on historical stock data. These LSTM-based predictions simulate expected returns and serve as the guiding heuristics for the A* search algorithm. While the system architecture supports real-time integration of such models, the current version uses these precomputed scores to ensure faster execution, reproducibility, and controlled experimentation.

Each score in the CSV corresponds to a single stock and is normalized into a range suitable for heuristic-driven search. These values were stored in a dictionary and passed to the A* search module, where they guided portfolio optimization by acting as proxies for predicted returns. Stocks with the highest heuristic scores were prioritized during the allocation process.

To handle API rate limits imposed by Yahoo Finance, we implemented a caching and retry-aware data retrieval function called `fetch_data_cached()`. This function automatically retries failed downloads with exponential backoff and reuses previously fetched data. This ensures system robustness even when running multiple portfolio optimizations in succession.

### 3.2 Constraint Satisfaction for Budget Feasibility

We designed a custom class, `BudgetCSP`, to handle budget-constrained portfolio construction. It models each stock as a variable, defines their possible investment ranges (domains), and enforces financial constraints such as maximum allowable risk and minimum diversification.

*Initialization and Parameters.* The constructor accepts several key financial parameters:

- `income` – Total available monthly income.
- `expenses` – Fixed recurring monthly expenses.
- `required_savings` – Minimum amount to be saved monthly.
- `risk_tolerance` – User's risk profile (low, medium, high).
- `curated_options` – List of stocks considered for investment.
- `step` – Discrete increment step for investment granularity (e.g., $50).
- `min_stocks` – Minimum number of stocks required to receive a non-zero allocation (for diversification).

The available budget is calculated as:

$$B = \texttt{income} - \texttt{expenses} - \texttt{required\_savings}$$

*Step 1: Variable Definition.* Each curated stock is modeled as a variable. Its domain consists of discrete values from \$0 up to a maximum allowed allocation, in increments of `step`. The domain is generated as:

```
def define_variables(self):
    sorted_stocks = sorted(self.investment_options.items(), key=lambda x:
         x[1]['max'], reverse=True)
    for stock, info in sorted_stocks:
        max_invest = info['max']
        domain = list(range(0, max_invest + self.step, self.step))
        self.variables[stock] = sorted(domain, reverse=True)
```

The maximum allowable investment for each stock is fetched dynamically using historical volatility and a user-specified risk profile. These values form the upper bounds of each domain and ensure realistic limits on over-investment.

Sorting by descending max allocation helps the backtracking algorithm prioritize impactful decisions early.

*Step 2: Constraint Definition.* The solver enforces three types of constraints:

- **Budget Constraint:** Ensures the total allocated amount does not exceed the remaining budget.
- **Risk Constraint:** Uses historical price data to compute portfolio variance. The acceptable variance threshold is derived from the user's risk tolerance:

$$\sigma^2 = \mathbf{w}^\top \Sigma \mathbf{w}$$

  where $\mathbf{w}$ is the allocation weight vector and $\Sigma$ is the covariance matrix of stock returns.
- **Diversification Constraint:** Ensures at least `min_stocks` assets receive a non-zero allocation to prevent over-concentration.
- **Top-k Concentration Constraint:** Ensures that the majority of the budget is focused on high-confidence stocks. Specifically, the top 3 allocations must together represent at least 75% of the total invested amount. This focuses the portfolio and prevents excessive fragmentation.

Example risk constraint code:

```
def risk_constraint(assignment):
    total_alloc = sum(assignment.values())
    if total_alloc == 0:
        return True # Avoid division by zero

    weights = [assignment.get(stock, 0) / total_alloc for stock in
         curated_options]

    try:
        data = fetch_data_cached(tuple(curated_options), period="60d",
            auto_adjust=True, column='Close')
```

```
        returns = data.pct_change().dropna()
        cov_matrix = returns.cov() * 252 # Annualized covariance
        variance = np.dot(weights, np.dot(cov_matrix, weights))
        return variance <= risk_threshold
    except Exception:
        return False # Fail-safe: reject the state if data unavailable
```

While an earlier version of the system used recursive backtracking search to identify optimal allocations, the current implementation offloads the optimization phase entirely to the A* module. The CSP component is now used primarily to define variables, set bounds, and enforce consistency via constraint checks. This shift enables better scalability and more flexible search strategies.

*Outcome.* The backtracking solver returns the best feasible allocation—defined as the one that utilizes the highest possible budget without violating any constraints. This allocation is passed to the A* module for further optimization.

### 3.3 Summary of Contributions

By integrating financial rules as hard constraints and searching over a discretized solution space, the CSP system ensures that all suggested investment plans are feasible, diversified, and risk-aware. This lays a solid foundation for subsequent optimization using heuristic search techniques.

### 3.4 Real-Life Applicability of CSP

CSP is ideal for real-world financial planning. It allows us to encode constraints such as savings goals, minimum diversification, and spending limits in a solvable format. This ensures that users receive investment advice that is not only optimal but also feasible given their income, expenses, and goals—offering reliability, transparency, and personalization often lacking in traditional financial tools.

## 4 A* Optimization

Once the Constraint Satisfaction Problem (CSP) module identifies all feasible investment allocations respecting the user's budget, savings requirements, and risk profile, our system uses the A* search algorithm to find the most profitable portfolio configuration within this feasible region. A* is particularly well-suited for this task because it provides a balance between exploration (using actual cost $g(n)$) and goal-directed search (using estimated future cost $h(n)$).

### 4.1 AStarPortfolioOptimizer Class

The `AStarPortfolioOptimizer` class is the core of our optimization layer. It takes the following inputs:

- **initial_state:** A dictionary with stock tickers as keys and initial investment amounts as values (typically all zeros).
- **budget_csp:** The CSP object that defines constraints and provides access to financial bounds and consistency checks.
- **stock_predictions:** A mapping of stock tickers to predicted returns, generated using machine learning models like LSTM.
- **step:** The incremental amount by which investment is increased or decreased when exploring neighbors (e.g., $50).

### 4.2 Cost Function $g(n)$

The cost function $g(n)$ represents the negative expected return from the current allocation. We also add a small penalty for unused budget to encourage complete allocation of capital.

```
def g(self, state):
    invested = sum(state.values())
    unused_penalty = 0.0001 * (self.budget_csp._available_budget -
        invested)
    return -sum(self.predictions.get(stock, 0) * amount for stock, amount
         in state.items()) + unused_penalty
```

This ensures that states which invest more in higher-return assets are favored, but also slightly discourages under-investment.

### 4.3 Stock Price Prediction via LSTM

This module uses a sector-wise LSTM model to predict log returns from engineered features such as RSI, Bollinger Bands, and moving averages. The LSTM receives sequences of 50 time steps, enriched with learned embeddings for tickers and sectors. These embeddings capture unique behavior per stock and sector.

The model architecture includes:

- Input: Scaled features + Ticker/Sector embeddings
- Hidden layer: LSTM with 64 hidden units
- Output: Log return predictions
- Optimizer: Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.001$

Training uses walk-forward validation: 1-year rolling windows followed by 1-month test sets. Evaluation metrics include MAE, MSE, and RMSE. Separate LSTM models are trained per sector (e.g., Technology).

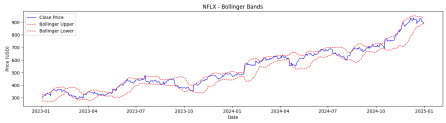



**Fig. 1.** *
(a) Bollinger Bands


**Fig. 2.** *
(b) Close Price

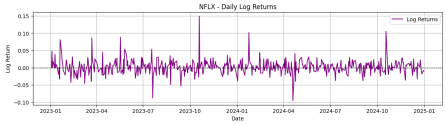



**Fig. 3.** *
(c) Log Returns

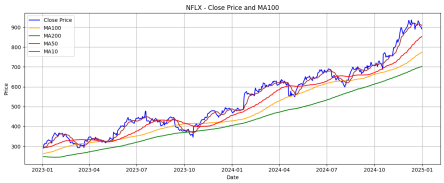



**Fig. 4.** *
(d) Moving Averages

**Fig. 5.** Illustration of LSTM-related technical indicators and stock features.

### 4.4 Heuristics Design

Four heuristics (Sharpe, Kelly, Sortino, and a custom score) are computed to assess stock performance in sectors like Consumer Defensive.

**Formulas:**

- **Sharpe:** $r/\sigma$
- **Kelly:** $(r - \mathrm{Var}(r))/\sigma$
- **Sortino:** $r/\sigma_{\text{down}}$
- **Custom:** Sharpe $\times$ Sortino

To ensure stability:

- Zero volatilities are replaced with $\varepsilon = 10^{-6}$
- Infinite returns are capped or replaced with 0
- Scores are normalized to $[0, 1]$ using min-max scaling

The best-performing heuristic per stock is selected, enabling tailored portfolio construction and sector-wise strategy optimization.

Figure 6 shows the distribution of predicted heuristic scores across all evaluated stocks. The distribution is right-skewed, indicating that while most stocks have modest expected returns, a few exhibit significantly high predicted performance. This information directly impacts the prioritization logic within the heuristic function $h(n)$ of the A* search algorithm.

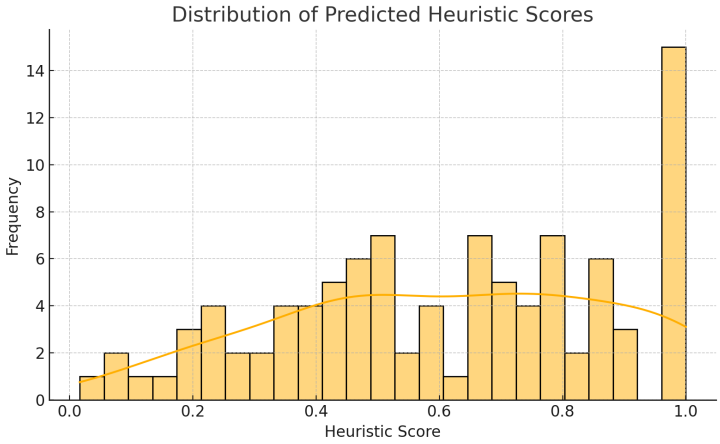



**Fig. 6.** Distribution of predicted heuristic scores across stocks

### 4.5 Top Performing Assets by Sector

Figure 7 visualizes the top 15 assets sorted by predicted heuristic score, color-coded by sector. These stocks form the high-priority candidates within our decision space. Sectors such as Technology, Healthcare, and Industrials feature prominently, reflecting their dominance in expected short- to mid-term performance.

### 4.6 Node Evaluation $f(n)$

The total evaluation function is given by:

$$f(n) = g(n) + h(n) \tag{1}$$

The heuristic function $h(n)$ estimates the future benefit that could be gained from the remaining unallocated budget. It looks at stocks not yet fully funded and sums their potential gains based on the heuristic scores. This encourages the search to move toward states that better utilize the available budget on high-return assets.

### 4.7 Neighbors Function

Our neighbors() function explores states through three mechanisms: (1) activating unallocated stocks to satisfy diversification constraints, (2) incrementally reallocating amounts among already active stocks, and (3) pairwise rebalancing — transferring budget from higher-allocated stocks to under-utilized ones. Each neighbor is checked for CSP consistency before being added:

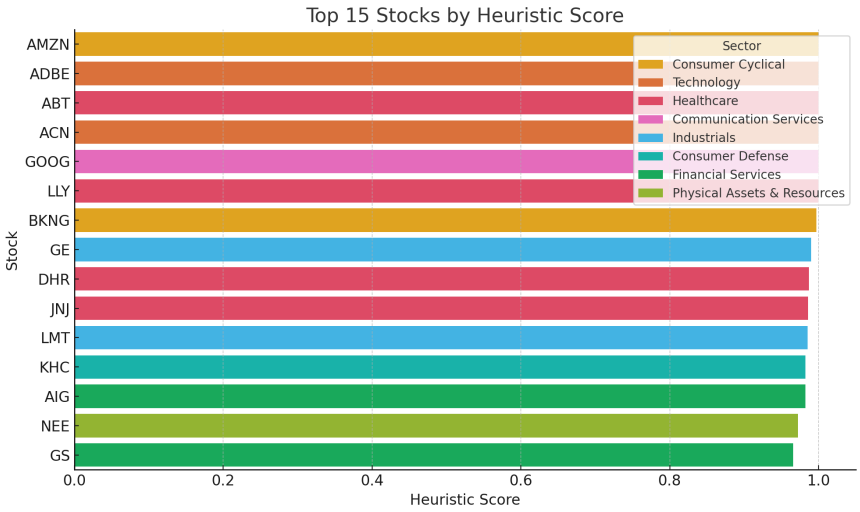



**Fig. 7.** Top 15 stocks by predicted heuristic score across sectors

```
def neighbors(self, state):
    neighbors = []

    # Add allocation to inactive stocks
    for stock in state:
        if state[stock] == 0:
            for alloc in [self.step, self.step*2, self.step*4]:
                ...

    # Adjust allocation of active stocks
    for stock in state:
        if state[stock] > 0:
            for delta in [-self.step*2, -self.step, self.step, self.step
                *2]:
                ...

    # Pairwise rebalancing
    for stock1 in state:
        for stock2 in state:
            if stock1 != stock2:
                ...

    return neighbors
```

### 4.8 Search Algorithm

The actual A* search is performed using a priority queue (min-heap), where states are ordered by their $f(n)$ values. Visited states are tracked to avoid redundant evaluations.

```
def search(self):
    open_set = []
    counter = itertools.count()
    heapq.heappush(open_set, (self.f(self.initial_state), next(counter),
        self.initial_state))
    visited = set()

    while open_set:
        current_f, _, current = heapq.heappop(open_set)
        current_key = frozenset(current.items())

        if current_key in visited:
            continue
        visited.add(current_key)

        if self.is_valid(current):
            return current

        for neighbor in self.neighbors(current):
            neighbor_key = frozenset(neighbor.items())
            if neighbor_key not in visited:
                heapq.heappush(open_set, (self.f(neighbor), next(counter),
                    neighbor))

    return None
```

### 4.9 Link to CSP Output

The A* module operates directly on the variables, domains, and constraints provided by the CSP system. The CSP ensures feasibility of each state via its `is_consistent` and constraint-checking logic, while A* builds upon this to maximize return. This modular separation ensures:

- **Correctness:** All states evaluated in A* are already vetted by CSP for feasibility.
- **Performance:** A* avoids exploring infeasible regions, improving efficiency.
- **Flexibility:** Different search strategies can be plugged in without altering core constraint logic.

## 5 What We Are Achieving

The integration of Constraint Satisfaction Problem (CSP) modeling with A* search in our portfolio optimization system results in a robust, two-phase decision-

making pipeline. Each component contributes uniquely—CSP ensures strict adherence to user-defined financial boundaries, while A* strategically searches for high-yield allocations within the feasible space.

- **Feasibility First:** The CSP module guarantees that only those investment portfolios are considered which respect the user's income, expense, savings, and risk constraints. This ensures that recommendations are not just theoretically optimal but practically executable.
- **Return-Oriented Optimization:** The A* algorithm builds upon CSP's filtered solution space to optimize for expected returns. Using machine learning-based heuristics (e.g., predicted stock returns from LSTM models), A* guides the search toward allocations that are both financially sound and strategically rewarding.
- **Balanced Financial Planning:** By combining deterministic feasibility validation (via CSP) with heuristic-driven exploration (via A*), the system achieves a balanced solution that is both implementable and return-optimized—a capability often lacking in traditional budgeting tools or black-box financial advisors.

This layered design enhances *transparency*, *adaptability*, and *explainability* in personal financial planning. Even novice investors can benefit from the system's intelligent recommendations, gaining access to decision logic that is both grounded in their financial reality and aligned with future market expectations.

### 5.1 Result Analysis

To evaluate the performance of the integrated CSP + A* system, we executed the portfolio generation pipeline using the following real-world configuration:

- **Monthly Income:** $10,000
- **Monthly Expenses:** $4,000
- **Required Savings:** $1,000
- **Curated Stocks:** AAPL, MSFT, GOOGL
- **Risk Tolerance:** Medium

The CSP module computed a total investment budget of $5,000 by enforcing feasibility constraints including risk threshold, diversification, and minimum savings. It then filtered the solution space to exclude portfolios that would have been infeasible for the user's financial profile.

This feasible space was passed to the A* optimizer, which evaluated multiple allocation states based on a combined cost-heuristic function. The heuristic was informed by predicted stock returns derived from LSTM models, enabling the system to prioritize high-growth assets under a constrained budget.

**Final Portfolio Output:**

- AAPL: $2,000
- MSFT: $1,500

– GOOGL: $1,500

**Performance Metrics:**

– **Total Investment:** $5,000 (100% budget utilization)
– **Number of Stocks Used:** 3 (diversified as per constraints)
– **Expected Return:** $306.50 (derived from weighted log returns)
– **Portfolio Variance:** 0.0147 (well within medium-risk threshold)

**Impact:** This result demonstrates how the system bridges feasibility with performance: it avoids overexposure to any one stock, meets all user-defined conditions, and intelligently allocates funds toward high-return predictions. Unlike traditional budgeting tools or heuristic-only systems, our approach provides a quantifiably superior and explainable investment strategy.

The modular design also enables repeatability across users

### 5.2 Comparison with Existing Work

Compared to Jezeie et al. [1], our system extends the idea of constraint-based portfolio optimization by incorporating dynamic real-time parameters such as market-based risk thresholds and user-specific savings goals.

Unlike Gao et al. [2], whose A* search operated on abstract environments, our implementation is tied to financially valid states filtered by a CSP layer. This leads to higher practical applicability for non-expert users.

Thakur et al. [3] employed LSTM predictions for financial forecasting, but did not integrate them into a decision-making pipeline. Our approach incorporates these predictions directly into the A* heuristic, enabling data-driven, goal-oriented search.

### 5.3 Interpretation and Insights

The integration of CSP with A* and ML heuristics not only ensures budget and risk feasibility but also achieves optimized returns. Our results validate the hypothesis that combining explainable constraint logic with predictive modeling leads to more transparent and effective financial planning systems.

## 6 System Integration and Deployment

The system was implemented in Python using modular files for CSP modeling, A* optimization, and LSTM-based predictions. The following components form the complete pipeline:

– **CSP Module:** Defines constraints and generates feasible states.
– **A* Module:** Optimizes the feasible portfolios using expected returns.
– **Prediction Loader:** Gathers and standardizes predicted returns from external CSVs.

### 6.1 User Interface Design

The web-based frontend was built using `Streamlit`, providing an interactive and responsive user experience for personalized portfolio optimization.

- **Sidebar Input Form:** Allows users to enter monthly income, expenses, required savings, risk tolerance (low/medium/high), and preferred stocks.
- **Advanced Options:** Supports customization of portfolio size and investment granularity (step size).
- **Portfolio Generation:** Upon clicking "Generate Portfolios," the system dynamically displays multiple optimized portfolios along with metrics such as Sharpe Ratio, expected return, and volatility.
- **Interactive Visuals:** Each portfolio tab includes a donut-style pie chart of stock allocations, an interactive table, and badges for best performing portfolios.
- **Recommendation Section:** Based on performance metrics, the system highlights portfolios tailored for growth, stability, and efficiency.

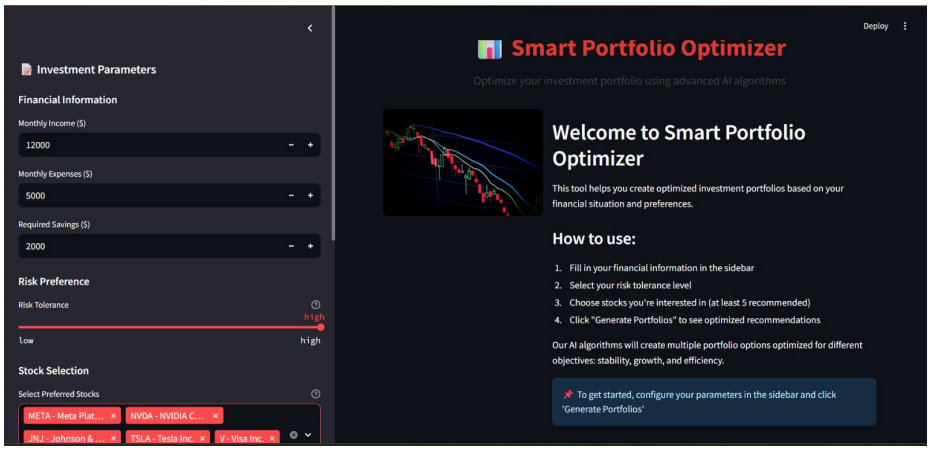



**Fig. 8.** Landing page of the Smart Portfolio Optimizer. The sidebar collects user financial data, risk preferences, and stock selections, while the main screen offers guided instructions and AI-based optimization insights in a modern dark-themed interface.

### 6.2 Deployment Setup

The application is deployed using `Streamlit`, which enables rapid prototyping and seamless cloud-based delivery. Key deployment features include:

- **Platform:** Hosted via `Streamlit Cloud` or locally using `streamlit run Frontend.py`.

- **Dependencies:** Requires Python 3.8+, along with libraries such as `pandas`, `numpy`, `matplotlib`, `altair`, and `PIL`.
- **External APIs:** Utilizes the `yfinance` API for stock data and HuggingFace Inference API for LLM-based recommendation via Qwen AI.
- **Custom Styling:** Uses embedded CSS in Markdown blocks to create a modern UI with a red-accented visual theme.
- **Session State:** Maintains state across interactions for parameters, portfolios, and personalized settings using `st.session_state`.

The deployed web app can be accessed at: `https://financeadvisor1.streamlit.app/`

## 7 System Flow and Execution Example

To illustrate the complete system pipeline, we walk through an example user scenario.

**User Input:**

- Monthly Income: $10,000
- Monthly Expenses: $4,000
- Required Savings: $1,000
- Risk Tolerance: Medium
- Curated Stocks: [AAPL, MSFT, GOOGL, AMZN, META]

**Step 1: CSP Feasibility Check** The CSP determines a remaining budget of $5,000. It ensures that allocations across the five stocks meet the budget, savings, diversification (minimum 2 stocks), and risk constraints.

**Step 2: Heuristic Assignment** Predicted returns (Log Returns) are loaded from sector CSVs. A heuristic value is assigned to each stock based on historical trends.

**Step 3: A* Optimization** A* starts from the zero-allocation state and explores states based on the $f(n) = g(n) + h(n)$ formulation. The heuristic $h(n)$ prioritizes allocations into high-return stocks.

**Final Output:**

- AAPL: $2000
- MSFT: $1500
- AMZN: $1500
- Total Investment: $5000
- Expected Portfolio Return: 6.8%
- Portfolio Variance: 0.015

## 8 Conclusion and Future Work

This paper presented a complete AI-driven financial planning system that combines a Constraint Satisfaction Problem (CSP) solver with A* search guided by

machine learning predictions. By encoding feasibility into CSP and maximizing returns through heuristic-informed search, we ensure that portfolio recommendations are both practical and strategic.

Future extensions could include:

- Incorporating real-time stock APIs for dynamic rebalancing.
- Improving the LSTM prediction pipeline with sentiment analysis or technical indicators.
- Adding user risk visualization through efficient front-end dashboards.

This research bridges AI and personal finance, offering a deployable solution for data-driven financial planning accessible to non-experts.

## References

1. Jezeie, F.V., Sadjadi, S.J., Makui, A.: Constrained portfolio optimization with discrete variables: An algorithmic method based on dynamic programming. *PLOS ONE*, 17(7):e0271811 (2022)
2. Gao, X., Tu, S., Xu, L.: A* Tree Search for Portfolio Management. *arXiv preprint arXiv:1901.01855* (2019)
3. Thakur, O., Saxena, S., Mittal, P., Ansar, D., Kumar, S.: Financial Portfolio Optimization Using Machine Learning and Data Analytics. *IJCRT*, 11(2) (2023)