High-Performance Machine Learning for FinTech

Anonymous authors
Paper under double-blind review

Abstract

The use of machine learning techniques in the FinTech sector, such as in portfolio selection, requires a high-performance compute engine to test a large number of candidate strategies. The higher the runtime performance of the engine, the larger the number of strategies that can be considered and the wider the scope of testing that could be conducted. This leads to better quality strategies. Thus, runtime performance should directly impact strategy performance in the market.

This paper presents a compute engine for processing market data and implementing highly customisable trading strategies. We use it as the core of our differential-evolution-based machine learning framework for portfolio selection. We then discuss a range of key techniques that improve its runtime performance and show that it outperforms other extant engines.

1 Introduction

At the heart of a successful machine learning engine for FinTech, there needs to be a high-performance compute engine to assess candidate strategies. The power of modern approaches to artificial intelligence (AI) and machine learning (ML) stems from the high number of ML models that can be tested within a short time. These models can be based on either neural networks or genetic programming. We consider Differential Evolution as an exemplar of the latter. Both approaches provide considerable flexibility, such as in the parameters of the neural network or the mutation and crossover procedure of the genetic framework.

In the context of FinTech, data analysts face two particular challenges. The first is the high volume of data that needs to be repeatedly processed, and the second is the complexity of the (investment) strategies that should be considered. We tackle the first challenge by developing a highly optimised compute engine, and tackle the second challenge by formally defining a rule-based expression language. This paper focuses on the compute engine but mentions key concepts of the expression language as a domain-specific language for FinTech.

The primary contribution of this paper is the systematic evaluation of the computational performance of our compute engine in comparison with other accessible platforms that offer ML technologies for FinTech. In discussing the core structure of the engine, we analyse the contribution to the performance of several optimisations in different components of the engine.

2 Context and Background

Prior research, such as Kumiega & Van Vliet (2012), has identified a research gap regarding algorithmic trading strategy development processes. Algorithmic trading has been increasingly dominating trading in financial markets for more than a decade. In this context, the drivers of performance are not individual decisions made by traders but long-term decisions made by managers regarding investing in individual tools for automation. For example which types of strategies should be researched further (checkpoints for profitability and robustness), or which can be improved for live trading (execution, specialised user interface, automating special cases)? Developing an individual strategy or feature might bind a developer resource for weeks or months. Without the ability to measure decisions at that level of abstraction, people are prone to form biased estimates of probability. For example, a strategy might have made the impression that it

is profitable due to data mining bias or wrong assumptions in a backtest. An interesting contribution can be made by providing development project managers and researchers with a tool to test the *robustness* of encoded processes. Encoded here means making economic theories specific enough so that they can be executed by a machine, tested by a tool, and their results quantified. Otherwise, money is maybe allocated to inefficient processes that lead to false-positive results that cause unstable or negative returns. Enhancing awareness of the process will help allocate resources (money and time) more efficiently.

Machine learning (ML) applications in business and research have increased in popularity. This is due to its enhanced pattern detection on large streams of data that were previously only accessible through the manual labour of data analysts. Today data analysts have not been replaced but empowered by novel ML techniques facilitated by computational advances. Evolutionary computing approaches have also been identified, e.g., by Bose & Mahapatra (2001), as a robust search technique for patterns in noisy data in large problem spaces, as long as the data format is uniform, which is common in finance. Although ML still faces regulatory and acceptability challenges (Lui & Lamb, 2018), its techniques are favoured in finance because of their transparency and ease of understanding by human supervisors (Wall, 2018). Genetic programming produces expressions in a grammar that is readable by humans. This should benefit the acceptability of the tool. Thus, the literature supports the idea that genetic programming (Lohpetch, 2011) is suitable for simulating strategy development. There are also other techniques like deep (Bao et al., 2017) neural networks (Ghazali, 2007), decision trees, clustering, classification, support vector machines, hidden Markov machines, and many more. See Wu et al. (2006); Aronson & Masters (2013); Luo & Chen (2013); Hargreaves & Mani (2015); Nguyen (2018); Arnott et al. (2019) for details. From our perspective, these techniques will also benefit from faster simulation speeds. They can be combined with genetic programming to create hybrid machine learning approaches. Genetic programming can either use inputs from technical analysis (Colby, 2002) or features extracted from other machine learning algorithms. It can also be used for the meta-optimisation of other machine learning algorithms. That is why we first focus on genetic programming and develop an expression language suitable for modelling hybrid machine learning approaches.

The problem with ML (similar to manual strategy development) in financial markets is that exploitable market inefficiencies could be short-lived in forward trading, and their detection can be subject to various biases. Aronson (2011) points to biases such as the data-mining bias, selection bias, survivorship bias, and curve-fitting bias, and how to mitigate them using a scientific process. The markets continuously adjust and thus an approach that is adaptable to changing market conditions is needed to verify the robustness of such an automated strategy development process. To do this, one has to apply automation concepts in a walk-forward manner, as defined by Pardo (2011), in the form of an automated longitudinal study.

Initially, we reviewed literature that discusses the choice of platform to use for backtests in financial research, or those that use particular platforms. This covers the usage of general purpose scripting languages such as Python (de Prado, 2018; Jansen, 2020) or Julia (Danielsson, 2011), statistics platforms such as Matlab (Chan, 2017) or R (Georgakopoulos, 2015; Conlan, 2016), and dedicated trading and more general FinTech platforms such as TradeStation (Pardo, 2011; Ehlers, 2013), MetaTrader (Blackledge et al., 2011), NinjaTrader (Ford, 2008), FXCM Trading Station (Mahajan et al., 2021), and Zorro (Liang et al., 2020).

We specifically focus on event-driven backtesting engines, but also compare against some vector-based backtesting engines. Vector-based strategy implementations are not well suited for live trading (sometimes requiring a rewrite), make alignment of multiple data feeds harder and are more prone to 'look-ahead' bias during research.[1] Matlab, R and Python have similar performance according to (JuliaLangContributors, 2022). However, the Julia benchmarks might also be biased according to Vergel Eleuterio & Thukral (2019) which shows better performance with Python in some cases. The Java benchmark is also biased because it uses a Java Native Interface (JNI) integration for a C-based matrix library. JNI adds significant overhead to a function call and the test measures mostly the performance of the C-binding of the matrix library. This is adequate when comparing matrix operations where Matlab, R, Python and Julia have their strengths, but they may not be the best-performing solutions for financial backtests. These general-purpose programming languages and trading platforms do not provide off-the-shelf solutions for generating strategies. Accord-

[1] The 'look-ahead' bias results in unrealistically high profits due to accidental misalignment of data that allows decisions to be influenced by future data points that will not be available under realistic assumptions (without a time machine that might not be physically feasible).

ingly, we also consider specialist platforms. StrategyQuantX and Adaptrade are examples referenced in the literature (see Hafiiak et al. (2018) and Hayes et al. (2013)). We also consider commercial tools, such as GeneticSystemBuilder (Zwag, 2020) and BuildAlpha (Bergstrom, 2020) because we find these to be faster than alternatives.

Extant available tools, whether commercial or not, do not offer the computational performance or the capability to automate robustness tests on strategy development processes. In the following sections, we will fill some research gaps regarding high-performance backtesting engines. This leads to an improvement in professional practice and empirical research by making viable test scenarios that were previously costly to implement in terms of processing power, manual labour, or time. Our simple domain-specific expression language and highly optimised compute engine enable the exploration of state-of-the-art ML technologies in the domain of algorithmic trading and portfolio management.

3 A High-Performance Platform: Invesdwin

To improve the current capabilities for empirical research on strategy development processes in economics, the first author has implemented a novel platform in Java. Over the last few years, this platform has been extended to support high-performance machine learning with a high degree of automation. We describe here the process of how a user can analyse strategy development processes in the platform. We also explain important architectural and design aspects to achieve high performance. Figure 1 shows the main structural features and processes of the Invesdwin platform.

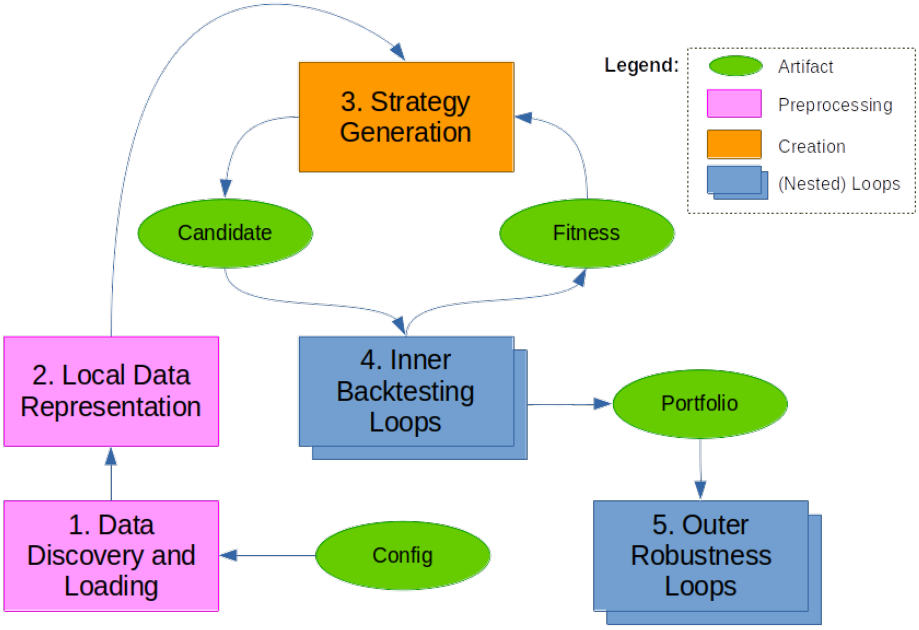


Figure 1: Strategy generation context and architecture of our platform

The platform can perform test scenarios for automated strategy development processes. Strategies are generated for specific markets. Individual candidate strategies are tested for profitability. They are then selected to form a *portfolio of strategies* to be traded forward. In the end, the robustness of the whole process is quantified in a Monte Carlo simulation. The platform, as shown in Figure 1, executes this in the following steps:

1. **Data Discovery and Loading** loads the required data from registered data sources based on the test configuration provided by the user. Depending on the deployment decisions, the data sources

can be embedded or accessed from a microservice infrastructure. Importantly for performance, data is cached locally in a specialised, low-latency, custom-developed NoSQL database that we designed specifically for this purpose (<anonymised>, 2022d). The processing of the tests can be done locally or parallelised in a distributed grid or cloud computing infrastructure. In the latter case, Steps 2, 3, and 4 below are executed on remote computation nodes, while Step 5 is aggregated and performed on the user's client computer. For this paper, we stick to a single computer though. As a novel technical contribution, high-performance and low-latency communication are facilitated by a channel abstraction that creates zero-copy and zero-allocation data pipelines (<anonymised>, 2022c). This step facilitates fast startup times for test scenarios without having to manually manage and copy large financial data sets.

2. **Local Data Representation** transforms the loaded data into an efficient in-memory structure (see Section 3.1). This extracts information and features from the data by using domain knowledge and machine learning approaches. The results are stored in primitive arrays of double values for indicator blocks (decimal results) and bit sets for signal blocks (boolean results). This step provides fast in-memory backtesting speed that is optimised for making heavy use of CPU prefetching without requiring slow data retrievals on the disk or network.

3. **Strategy Generation** defines and combines expression blocks (indicators and signals) for creating candidates for trading strategies. This is done by finding local optima in the problem space by using machine learning (specifically differential evolution for this paper). The individual fitness of the candidates is evaluated here by checking the trade frequency, profitability, and other metrics. Our highly optimised expression language (<anonymised>, 2022b) and many semantic shortcuts are implemented here to walk through the problem space as efficiently as possible.

4. **Inner Backtesting Loops** evaluate the *robustness of individual candidates* and filter them into portfolios by using domain-specific heuristics and machine learning. These portfolios are then used in an outer walk-forward (Step 5) that evolves the portfolios in regular intervals. Thousands of profitability/success metrics are made available in a way that does not cause a significant overhead for storage or computation. With profitability/success metrics we not only mean Profit/Loss, Sharpe Ratio, or Drawdown but also complex statistical tests that look at the complete backtest history. This is essential when testing hundreds of thousands of generated strategies per second.

5. **Outer Robustness Loops** evaluate the **robustness** of the strategy generation and portfolio selection approach. Besides entries & exits (entering a trade and exiting a trade by sending orders to the broker based on rules) or portfolio management (e.g. selecting markets and strategies regularly, maybe based on fitness criteria), more decision points such as risk management (e.g. portfolio/market/industry/direction/exposure-based loss limit), position sizing (e.g. volatility based or equal risk per cent), and equity curve trading (temporarily stop trading after certain losses in a day, or decouple a strategy from live execution to simulation during drawdowns in isolated strategy equity), can be automated as well. This allows the testing of hypotheses on formalised strategy development processes and the making of decisions at higher levels of abstraction. The walk-forward runs of the inner backtesting loops (Step 4 in Figure 1) are repeated multiple times here (e.g. 200 times) to sample the distribution of out-of-sample results in a Monte Carlo simulation. The *strategy generation is randomised*, thus each run will have slightly different results. For example, based on fluctuations in metrics of profitability such as Profit/Loss or Sharpe Ratio. By comparing distributions of results from strategy development processes, we are then able to compare and rank the alternatives. This decision-making can be formalised, as we have done in our domain-specific expression language, and fully automated. We define a **robust strategy development process** as one that has a distribution with **low variance and high average profitability**. It does not become unprofitable over many trials and survives validation checks over multiple markets and time frames (as examples of approaches that can be used for validation in the platform). We can quantify robustness and automate decisions about them. This step is not available in extant platforms and is hard or impossible to formally research without having access to the processing speed and automation available in our platform.

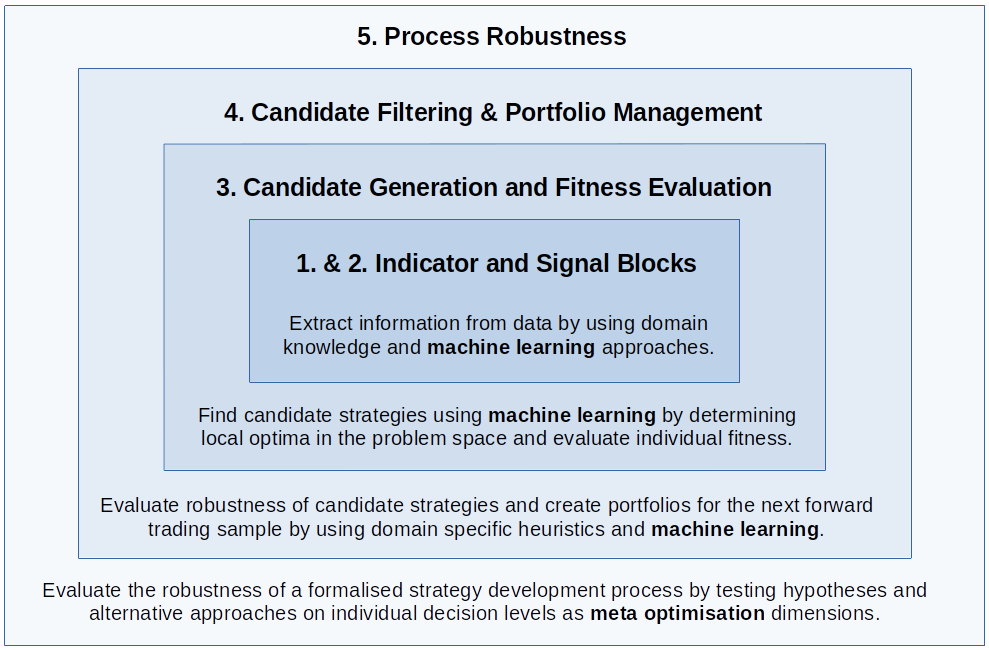


Figure 2: Machine learning layers

Figure 2 depicts the different levels of abstraction in this model as layers of machine learning. Hybrid machine learning approaches can be composed **horizontally** (architectural integration) to create a combined decision on the same layer, **vertically** (data manipulation) by connecting inputs and outputs across multiple layers, and on a **meta optimisation** (model parameter) dimension in one layer (see Anifowose (2020) and Anifowose et al. (2017)).

3.1 Backtesting Engines

A fast backtesting engine is the component that is most critical to performance for applying ML techniques because many alternatives need to be tested as fast as possible. Three possible types of local data representation are implemented as specialised backtesting engines with the main difference being a different storage format (for simulating trades of a strategy candidate based on historical data):

1. **Historical:** We define indicators as transformations of the price data to extract features. Signals are defined as true/false interpretations of indicators that lead to buy/sell/hold decisions. This engine stores indicators and signals as in-memory historical caches (<anonymised>, 2022a) during live trading to minimise memory usage while providing multithreaded access to indicators and signals for visualisation purposes in charts. Indicators and signals are calculated lazily in a pull-based interaction between them. Thus if a strategy does not need a specific calculation for a while, it can be skipped.

2. **Circular Buffer:** Another implementation might use circular buffers of primitive arrays as the in-memory storage of the indicators to speed up the calculations for low latency live trading or to make backtests of large data sets utilise a limited large memory space in a moving window. Tick data is the smallest granularity and highest volume of data points that can be processed. Multiple ticks can occur in a millisecond, and order book data can provide further information on 'depth'. An example would be to maximise the speed of portfolio tests on tick data, which would not fit into memory when multiple years are to be tested. Data might be calculated eagerly here via an observer pattern. This engine is still under development.

3. **Blackboard:** Another representation is to precalculate the whole time series into primitive arrays for indicators and signals and keep those immutable. This is stored in a memory space following the blackboard pattern (Buschmann et al., 1996, p. 71). This provides memoisation of indicators and signals during backtests (Mayfield et al., 1995). Individual backtesting threads can fetch or add more blocks on demand. Unused blocks are evicted automatically based on memory management heuristics (least recently used, soft & weak references, memory limits). This provides the fastest access for machine learning tasks where the whole time series is iterated over many times to find an optimised variant of a strategy that performs best.

From a computational performance point of view, we are interested in machine learning problems that use a form of genetic programming, specifically Differential Evolution, to generate and test a large number of strategy candidates. Thus the blackboard option provides the fastest internal representation of the workload. Using (and reusing) precomputed primitive arrays as the in-memory representation allows CPU prefetching in the hardware to be used extensively during the iteration of the backtests. Our column-oriented storage is preferable to row-oriented storage because it minimises cache misses by the CPU (Ragan-Kelley, 2014, p.34 ff.). Each column represents an indicator or signal, or another expression that memoises a function on multiple indicators or signals.

3.2 Expression Language

The platform uses a domain-specific expression language to formalise *building blocks* for strategy generation (indicators and signals) and to automate portfolio management and other high-level decisions (<anonymised>, 2022b). Indicators (decimals) can be combined into signal blocks (boolean) by comparing them against each other, or thresholds, in various ways as expressions. The expression language is rule-based with boolean guards that trigger actions: *"Rule := Guard → Action"*. Figure 3 shows examples of such expressions and the generation process:

Similar to our platform, BuildAlpha (2022) simplifies this by only allowing curated signal blocks in the strategy generation process. Only logical "and" operators are allowed for combining multiple blocks. The backtesting speed of BuildAlpha is the fastest extant tool (apart from ours) because rather than mathematical expressions it evaluates simple boolean expressions lazily via shortcutting. When one condition becomes false, the other conditions in a logical "and" combination can be ignored, and one does not have to calculate additional indicators. Since the signal in a trading strategy is false most of the time, this improves the backtesting speed significantly.

3.2.1 Expression Language Design

An excerpt from our language design document (<anonymised>, 2022b): "Our expression language is deliberately not Turing complete, and components need to be of limited resource consumption. It is designed as a domain-specific language that supports variables and functions. It supports no recursion, loops, arrays, lists, collections, maps, strings, exceptions, declarations, or assignments. Instead, non-recursive functions are used to emulate loops, mappings, decisions, and selections in a simplified fashion with a limited potential for coding errors. Functions can be nested arbitrarily as dynamic parameters. Thus a mathematical view of higher-order functions is supported as nested transformations on *double* series.[2] There is no notion of a functor, function object, or variable for a function in the language. Only the underlying implementation has that representation and is hidden from the user. Instead, the user can think in a series of *doubles* (data) which are provided and transformed by functions to be fed into other functions as parameters. This comes from the fact that the language operates on streamed financial data and can access previous data and calculation results. Since it operates solely on *doubles* and indexes, the function algebra is simplified significantly. The language makes the use of functions and variables transparent to the user by making parentheses op-

[2]For example, a *double* series consisting of {1.1, 1.2, 1.3} is transformed via nested transformations: "*multiply(round(0), 2)*" into a new *double* series consisting of {2, 4, 6}.

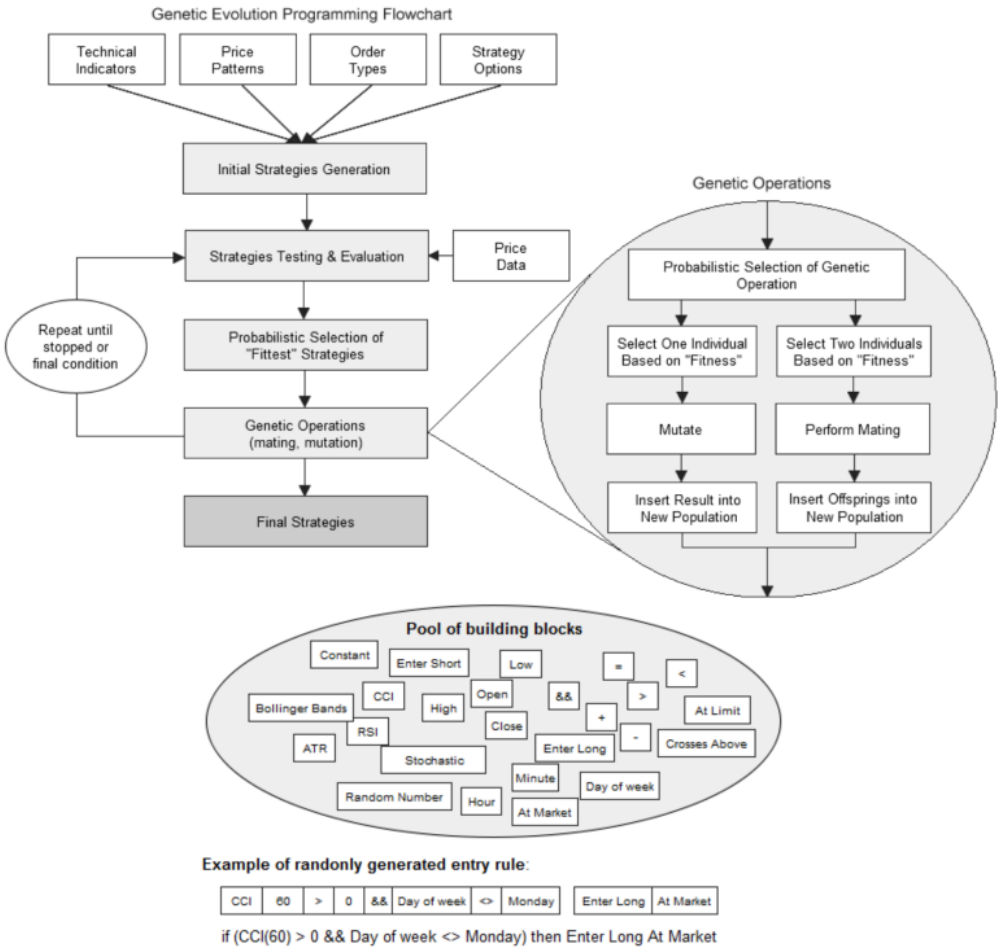


Figure 3: Genetic generation process and boolean expression example (StrategyQuant, 2022)

tional.[3] Typical mathematical and logical operations are supported. Constants are represented as variables in the language. The language is case insensitive. The language has only a single type *double* for inputs and outputs. Other types like *boolean* get encoded into *double* by seeing numbers above *0* as *TRUE* and all other values as *FALSE*. *Null* values are represented by the *double* value *NaN* which means "Not a Number". The underlying implementation may transparently use boolean, Boolean (object), integer, double and generic object types where this provides a performance benefit."

3.2.2 Signal Strategy Example

A strategy that uses simplified boolean expressions combines multiple signals that are composed of interpretations of indicators based on domain knowledge to decide about an entry point for a trade. Domain knowledge here means for example technical analysis or statistics based on literature (or ML models trained on the data). Another simplified boolean expression with an optional time/loss/profit-based stop handles the exit of the trade. Only one trade at a time is allowed. Such a strategy can be expressed schematically in a rule-based form, where a guard is a boolean expression that decides whether to trigger a (sequence of) actions:

[3]This is realized in the syntax. Variables may or may not be suffixed with parentheses. The formats "*variable()*" or "*variable*" are both accepted. Also, parentheses can be omitted for functions with only optional or no parameters. The formats "*function*" and "*function()*" are both acceptable.

- *Rule := Guard → Action*[4]
- **Entry :=** signal1 && signal2 → **enterLongAtMarket()**
- **Exit :=** with the following Guards and Actions:
  1. signal3 && signal4 → **exitAtMarket()**
  2. stopLoss(volatilityRange1) → **sendStopLossToBroker()**
  3. takeProfit(volatilityRange2) → **sendTakeProfitToBroker()**
  4. maxBars ≥ x → **exitAtMarket()**

An example strategy might look like this:

- **Entry :=** close[1] < close[0] && volume[1] < volume[0] → **enterLongAtMarket()**
- **Exit :=** with the following Guards and Actions:
  1. average(close, 10) < average(close, 20) && relativeStrengthIndex(2) < 0.7 → **exitAtMarket()**
  2. stopLoss(averageTrueRange * 0.5) → **sendStopLossToBroker()**
  3. takeProfit(averageTrueRange * 2) → **sendTakeProfitToBroker()**
  4. maxBars >= 5 → **exitAtMarket()**

An expression is evaluated for each data point in the time series individually in a backtest or as data arrives in live trading. This strategy will enter a market order profiting from an upward trend when the last bar closes above the previous bar (Entry, left) and the volume has increased (Entry, right). It will exit the trade when a fast moving average is below a slow moving average (Exit 1, left) and the Relative Strength Index is below 70% (Exit 1, right). Alternatively, the trade might also exit at a 50% volatility loss (Exit 2) or a 200% volatility profit (Exit 3). If neither of these exit conditions occur, the trade is closed after 5 bars (e.g. days) after the trade has been filled by the broker (Exit 4).

Other strategy generators are:

- Mathematical Strategy: Uses mathematical operators to create a decimal value that is compared against a threshold. This requires calculating everything without shortcutting. Since this type of generator is comparatively slow (see Section 5.4), we have not yet implemented it in our platform.[5]
- Breakout Strategy: Filters trade opportunities based on signals, then uses a mathematical expression to define a price target for a limit or stop order.[6] We have implemented this and can make use of our signal optimisations even though trade simulation is slower than in signal-only strategies.

3.2.3 Portfolio Selection Example

Candidate strategies can also be selected for a portfolio through a ranking and filter expression. We mean a portfolio of dynamic strategies on one or more instruments. Common portfolios only consider buy & hold investments in stocks, bonds, and funds. The schema might look like this:

[4]The actual syntax uses logical "and"/"or" to combine the guards with mostly implicit actions. This would be the syntax for entry: *"signal1 && signal2 && enterLongAtMarket()"*, and for exit: *"signal3 && signal4 || stopLoss(volatilityRange1) || takeProfit(volatilityRange2) || maxBars ≥ x"*. In the entry, *"&& enterLongAtMarket()"* can be omitted since this would be the implicit action (if configured for the strategy). It is useful to declare it explicitly for actions like limit/stop entries. Limit/stop orders can also be declared for both long and short positions simultaneously by OR combining them into one entry rule. This way, breakout strategies can be expressed. These are implementation details of the language that we express more simply with the rule-based notation.

[5]If we do implement it, we might find ways to accelerate it. This might enable us to find other kinds of alpha-capturing trading strategies than our current generators provide.

[6]An example breakout strategy can be schematically defined as: *"filterLong && enterLongAtStop(longPriceLevel + volatility * factor) || filterShort && enterShortAtStop(shortPriceLevel - volatility * factor)"*.

- **Rank & Filter :=** with the following Guards and Actions:
  - fitnessFilter1 && fitnessFilter2 → **removeCandidateStrategy()**
  - rankDesc(rankFitness) <= portfolioSize → **removeCandidateStrategy()**

An example expression might look like this:

- **Rank & Filter :=** with the following Guards and Actions:
  - tradesCount >= 100 && sharpeRatio > 0.8 → **removeCandidateStrategy()**
  - rankDesc(profitLoss / maxDrawdown) <= 10 → **removeCandidateStrategy()**

This expression will create a portfolio of the 10 best candidates based on the return-to-risk ratio which have more than 100 trades and a Sharpe ratio above 0.8.

3.3 Differential Evolution

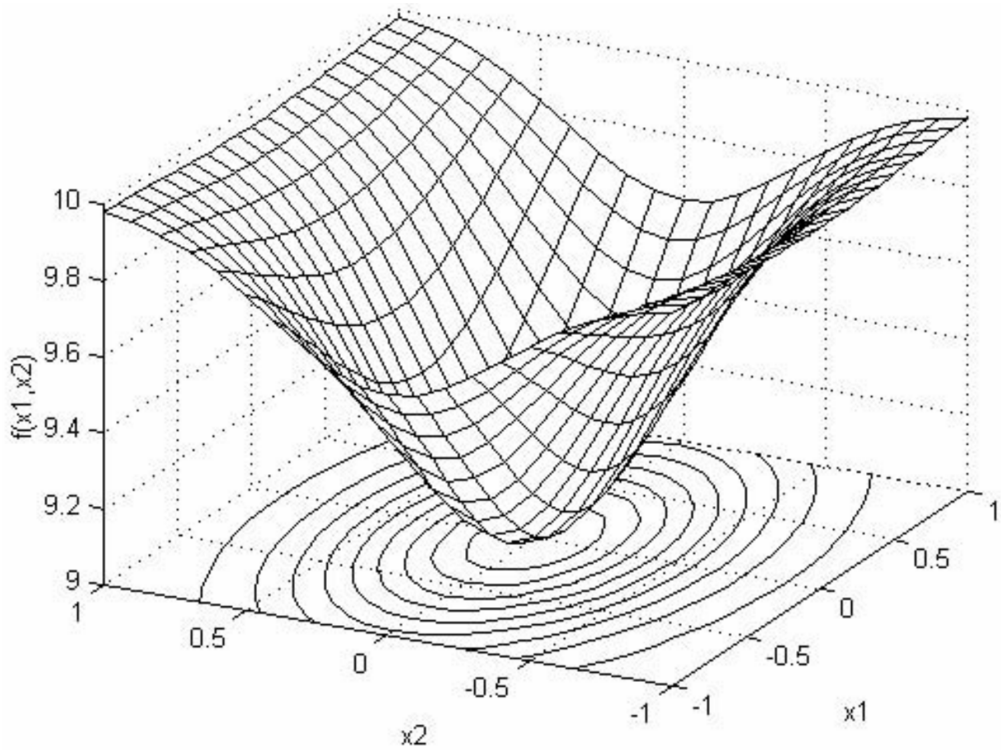
We use Differential Evolution (DE) (Storn & Price, 1997) to generate our strategy candidates. Other generators can be implemented and tested whether they converge faster to viable strategy candidates. We use DE as a good starting example.

3.3.1 Encoding Strategy Candidates

For a strategy that uses simplified boolean expressions with 2 "and" combined signal blocks for the entry rule and 2 "and" combined signal blocks for the exit rule, the problem would be stated as 4 integer variables from -1 to the number of different available blocks per variable. We use 100 blocks in our example and, thus, a maximum index of 99 since array indexing starts at 0. Each array represents a list of domain-specific rules that extract features from the market data. As an example, consider the following list of 100 rules:

- Index -1 disables this block.
- Indexes 0 to 20 are variants of moving averages and other long-term trend signals.
- Indexes 21 to 35 are volatility signals based on volume or price action.
- Indexes 36 to 70 are short-term momentum rules for mean reversion based on the Relative Strength Index and absolute momentum measures.
- Indexes 71 to 85 are time and session filters to restrict when signals are to be taken.
- Indexes 86 to 99 are Bollinger Bands and other channel-based signals for long-term breakout signals.

The problem space results in the permutations of the variables being multiplied. A strategy candidate picks 4 of the 101 potential blocks (including -1 for a disabled block and the rules from 0 to 99). It uses the first two variables for the entry signal and the last two variables for the exit signal. Figure 4 shows a two-dimensional problem space where the third dimension is the fitness value of negative Profit/Loss to minimize. The minimum to which DE should converge is apparent in the figure. The rings on the bottom are the contours that DE uses to move to the minimum. Typical problem spaces are more complex and look more like an inverted mountain region, or a crater region on the moon, with multiple local minima where DE might find suitable strategy candidates. However, with our 4 variables, we operate in a four-dimensional problem with the fifth dimension being the fitness value. Sadly this can not be visualised easily, but luckily DE can handle higher dimensions well because it treats it as a contour space that guides it.



(For better visualisation: Axes of input variables x1 and x2 are normalized between -1 and 1; f(x1,x2) as the fitness output is scaled between 9 and 10)

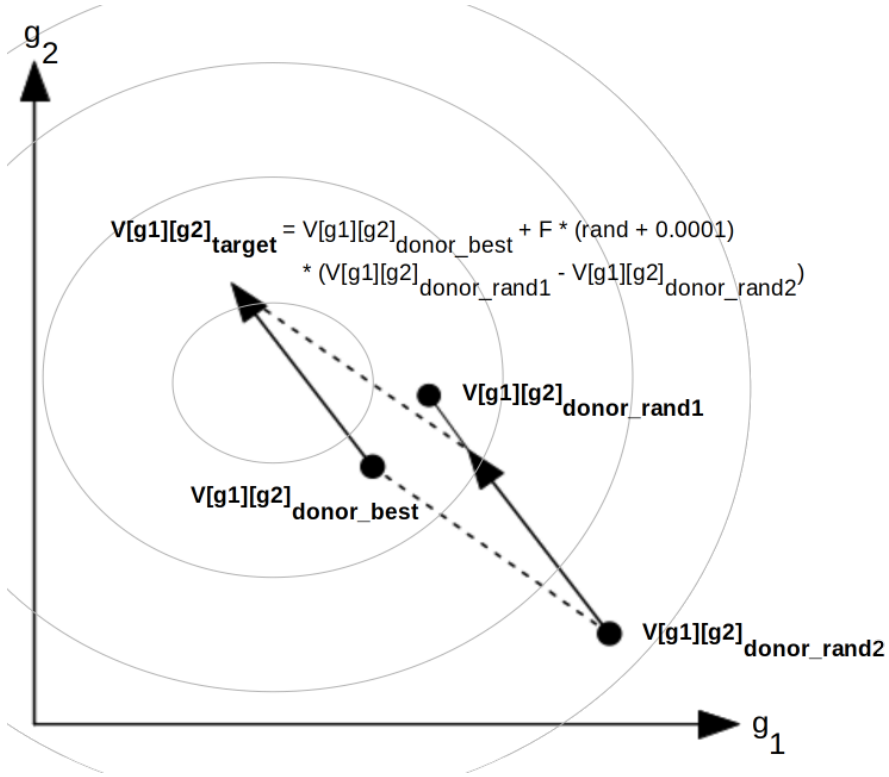
Figure 4: Two dimensional problem space with fitness as third dimension, adapted from (Price et al., 2005, p. 9)

3.3.2 Vector Movement for Crossover and Mutation

Since the rule arrays are sorted logically so that similar rules are near each other, an algorithm such as DE can benefit from directional movements in the contour space. Larger jumps in the indexes caused by mutations and crossovers will result in different kinds of rules (represented by individual inverted mountains or craters in the problem space), while smaller jumps will lead to similar rules that might differ in parameterisation only for fine-tuning (approaching the local minimum). Our generator, on which we base our benchmarks, uses 3818 domain-specific rules and includes an efficient negation of them that results in twice as many (7636) rules. This leaves enough space for smaller jumps to pick similar rule variants and enough local minima to find many strategy candidates. This is a bit harder to imagine, so we stick to 100 rules and a demonstration of a three-dimensional problem space in our explanations.

Figure 5 shows how DE calculates the vector movements (for each input that we call gene) in the problem space using the formula (at the top) for crossover and mutation (derived from Georgioudakis & Plevris (2020)).

Figure 6 shows how DE is executed. The algorithm starts with an initial population of 500 candidates generated randomly. The scaling factor is a constant value that moves based on a random value at a speed defined by the difference between two random candidate values at the same gene. Since we have integer values, we add a step that rounds the modified vector indices for the new trial candidates to the nearest integer. Since the speed can move the picked signal beyond the allowed values, we truncate the result between -1 and 99. When -1 is picked, the given block is disabled. This allows us to test simpler strategies with less than the maximum amount of signal blocks. In the implementation, we perform the Mutation and Crossover steps together as one step without actually creating an intermediate vector. This saves allocations and improves the speed of the algorithm. Also, we diverge from other DE implementations by modifying



(Instead of x1 and x2 as inputs, we speak of g1 and g2 as genes of the vector tuple V[g1][g2] that represents a candidate that is on the contour space)

Figure 5: Vector movement in the contour space, adapted from (Price et al., 2005, p. 39)

the population and then evaluating all together as a separate step instead of a tight loop that tests each candidate individually. This creates a bit more randomness in the search and allows for running backtests together depending on the engine.

The number of backtests can be reduced by sorting the picked blocks in the variables of a generated strategy candidate by the true count ascending. That way logically duplicate backtests can be filtered in advance via a score cache. If it is a new candidate, the backtests will be faster because the first block will most often decide not to enter a trade, thus the next blocks don't need to be checked most of the time. However, this optimisation has no effect if block compression (see Section 4.2) is enabled since that combines the blocks into one "bit set" upfront. Instead, it still applies to expressions that mix with dynamic blocks.

The parameters for the crossover probability P and the scaling factor F could be meta-optimised separately.

We execute the strategy generation in multiple threads which perform backtests for the evaluation of the DE algorithm. Each thread runs DE in a loop until the pre-configured number of unique and valid candidates has been generated. It can also abort with a smaller number of valid candidates if too many duplicate or invalid candidates are found. In addition, there are hard limits on the maximum number of candidates checked and this limits the maximum number of backtests and iterations. For this, DE runs multiple loops with a given population size and convergence parameters that define how the crossover and mutation should take place. Each evolutionary step attempts to improve the population until either a maximum iteration count is reached or the population is unable to improve over a few iterations. The best individual is kept between iterations, and candidates that did not change will not be backtested a second time. The best individual from the final population is then used as the strategy candidate for the outer portfolio selection. A shared fitness cache prevents the repetition of backtests for duplicate strategies between parallel strategy generators. Duplicate strategies are removed from the resulting candidates by comparing the final fitness

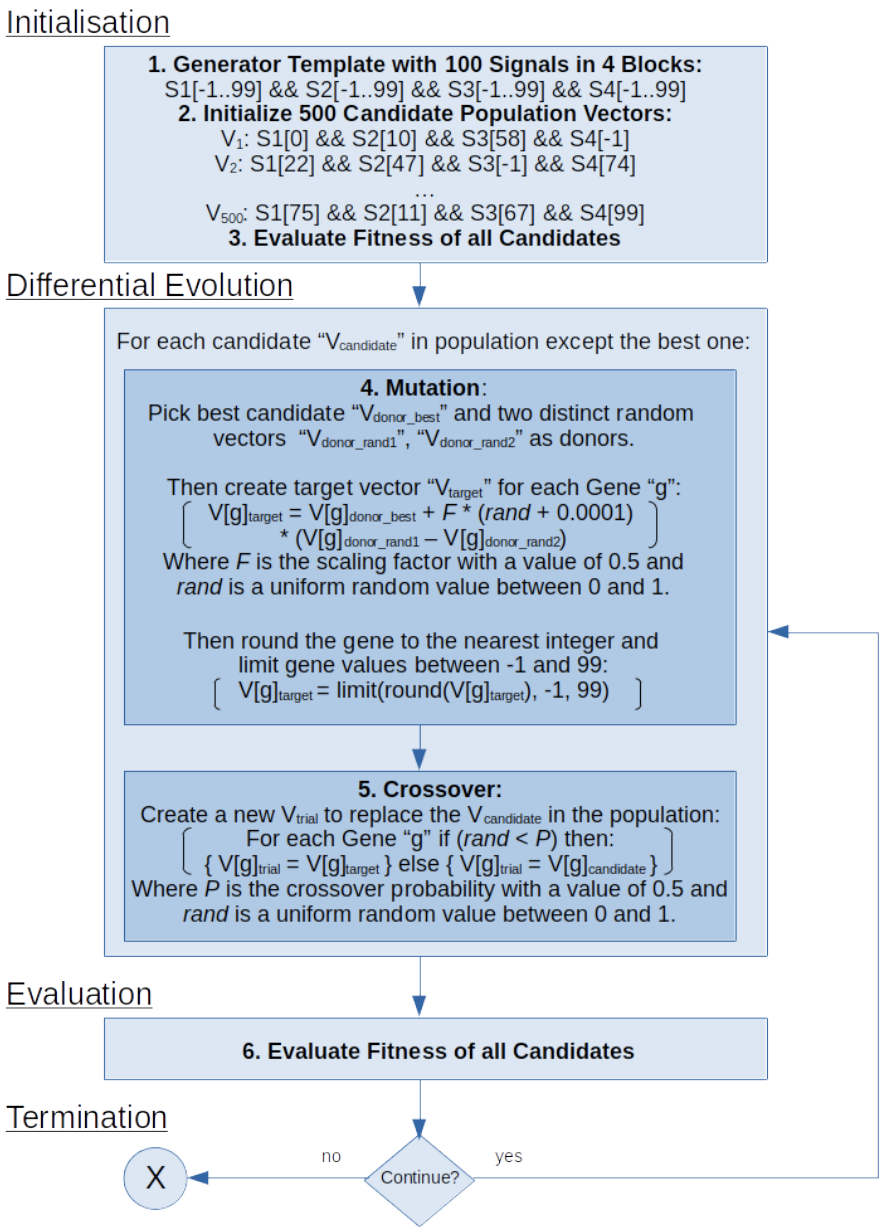


Figure 6: Differential evolution process

and the trade count against other candidates. Invalid candidates without trades are punished with the lowest possible fitness during the generation process, because DE would always favour no trading over a strategy that produces losses.

3.3.3 Nested Backtest Loops

Figure 7 shows multiple options to run nested loops as *Post-Optimisation* (in respect to strategy generation) inside each backtest for:

- *Nested Optimisation*: This runs a brute force optimisation (that explores all variants) for the given blocks to find the best parameters for each block. DE will see the best fitness value to make its

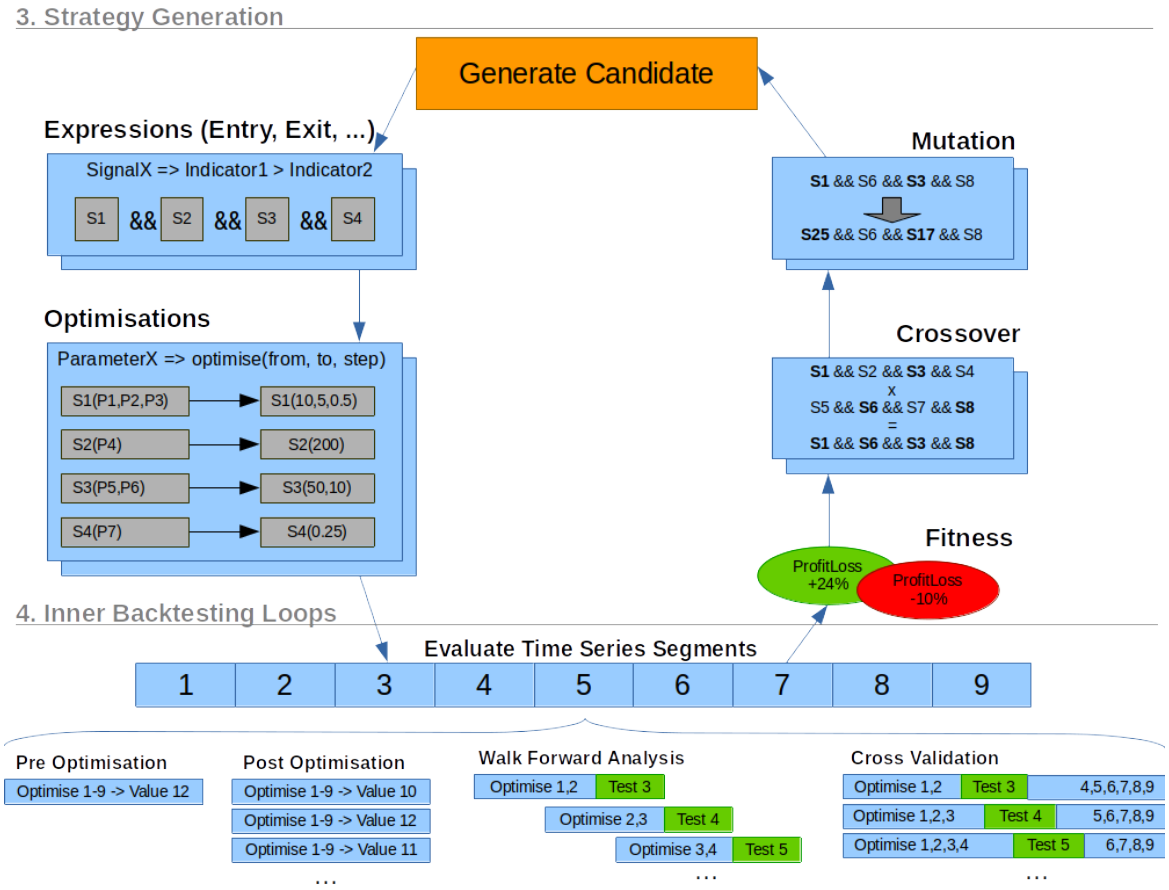


Figure 7: Candidate generation and evaluation process

decision about an individual. To limit the number of alternatives to be tested, one parameter is optimised after the other. This adds the possible permutations as tests by multiplying them during the optimisation process. Alternatively, a genetic optimisation algorithm could be used to reduce the number of tests when using multiplied permutations.

- *Nested Walk-Forward*: This repeats a brute force optimisation multiple times in a walk-forward manner and concatenates the out-sample walk-forward steps as the fitness value for an individual.

- *Nested Cross Validation*: This splits the testing period into multiple segments and repeats a brute force optimisation for each segment as the out-sample period individually while the other segments are used as the in-sample optimisation period. The out-sample fitness is then averaged over the out-sample segments for an individual.

There are also some options to select different optimisation parameters for blocks during the initial precalculation. These are called *Pre-Optimisation* options besides the *No-Optimisation* option which leaves the default values as they are. *Pre-Optimisation* can make full use of precalculated indicators and signals. It is more involved for *Post-Optimisation*. The platform has to calculate all permutations of a block and cache these inside the block so that values can be reused as primitive arrays within optimisation cycles. This requires significantly more effort to implement but allows high-performance advanced optimisation workflows that give DE a higher level of abstraction to decide about strategy candidates. Though for our comparison benchmarks we disable these kinds of nested loops because no other platform supports them. Instead, optimisations are flattened into multiple blocks for the same rule.

The fitness evaluation is done via deliberately simple and, consequently, fast algorithms that measure specific statistics about the strategies as fast as possible. In this manner, the nested loops only perform the minimum amount of calculations to decide about an individual. A strategy candidate that is given to the portfolio selection is re-evaluated with a more complex suite of statistics. This makes available all possible statistics for complex ranking and filtering logic that determines which candidates should be included in the portfolio. The complex suite of statistics calculates equity-based measures (Profit/Loss, Sharpe Ratio, etc.) on a daily basis and feeds the information into fast algorithms that are calculated eagerly. The same applies to statistics for orders (Win Percent, Z-Score for Consecutiveness, etc.) that are pushed eagerly. More complex and expensive algorithms (to check the robustness of candidates) are calculated lazily based on other statistics, the stored daily equity curve or individual trade events. Portfolio rules can also compare statistical properties like correlation or cointegration between candidates or apply advanced hybrid machine learning techniques at this step. With this mix of eager and lazy calculations, portfolio decisions can be made as fast as possible for a given configuration. The immutable results can be stored in files for later analysis using external tools or be used in the following analysis steps.

4 Key Contributors to Performance Improvements

In this section, we summarise the key optimisations that have been undertaken on the platform to achieve high-performance evaluation, to the extent that it exceeds that of other extant platforms according to the measures reviewed in Section 5 below.[7] The hardware is comprised of an Intel i9-9900k (8 physical cores, 16 virtual cores, 16 MB cache, from 3.6 GHz up to 5 GHz turbo boost) with 64 GB RAM and SSD storage. We use EUR/USD as the instrument with tick data sourced from DukascopyBankSA (2022b) including the bid/ask spread. One minute, four hour and daily bars are aggregated from this data. This is used for all benchmarks in this paper. We go through the performance contributors individually in the order of their importance.

4.1 Faster Simplified Boolean Expressions

Our platform can leverage similar speed improvements as BuildAlpha (2022) as well as add some novel optimisations. For example, logical signal blocks can be stored efficiently in *bit sets* (1 bit per value) instead of primitive boolean arrays (8 bits per value in Java). These bit sets use primitive long array types (64 bits in Java) where each bit is used to store a boolean value (1 bit boolean). This makes CPU prefetching significantly more efficient and requires 8 times less memory. Additionally, we can use bit set "and" operations to **compress** multiple signals into one bit set before a backtest. The backtest can then be skipped for bit sets that are never true or **skip** through segments that are false for a while (which is most of the time). The skip is implemented as a lookup in the bit set to find the next time index that requires the backtesting logic to execute a trade again. Thus, the full backtesting logic is performed when needed only. This is required for calculating trades but not for waiting on a trade signal. We call this technique **"Skipping Compressed Bit Sets"**. This is one of the main performance contributors that is benchmarked in Section 4.2.

4.2 Boolean Compression

Table 1 shows the performance contribution of different storage formats for signal blocks, as discussed in Section 3.1. This is the main performance contributor when two advanced techniques are applied. We call this 'compression and skipping with bit sets'. **Compression** combines multiple bit sets via a quick "and" combination into one bit set representation that can be iterated faster and knows how often and when the combined expression is true. Backtests can be skipped entirely when expressions are false over the whole time series. When this compression is not available, lazy evaluation via shortcutting of boolean combinations can **skip unneeded calculations**. But knowing how often and where the whole time series becomes true allows for a second advanced optimisation. Sparsely true combinations benefit from skipping pre-computed time series indexes within backtests by looking into the bit set for the next true index and continuing the

[7] These measures are processing speed on ticks and bars for backtesting and optimising classical and ML-generated trading strategies.

backtest evaluation from there. This skips complex calculations within the backtesting engine that would have arrived at the same conclusion but with a code path that is more expensive than a lookup in a bit set.

For automated strategy generation, we choose to generate strategies with 8 signal blocks based on simplified boolean expressions for the entry and a time-based exit after 1 bar. Entries and exits refer to entering a trade and exiting a trade by sending orders to the broker. In terms of testing frequency, we test on daily bars with about 20 years worth of historical data. We also show the effect of skipping to the next true value during backtests. We measure the backtests per second and extrapolate the bars per second from that.

Table 1: Boolean storage format performance overhead

Optimisation	**Bars**	**Backtests/s**	**Bars/s**	**Relative**
RoaringBitmap	5,839	18,439.99	107,671,101.60	-47%
BitSet	5,839	34,367.77	200,673,409.00	-1.3%
Boolean Array (Primitive)	5,839	34,832.96	203,389,653.40	Baseline
Boolean List (Object)	5,839	35,379.05	206,578,273.00	+1.6%
Compressed RoaringBitmap	5,839	49,779.17	290,660,573.60	+42.9%
Skipping RoaringBitmap	5,839	51,289.57	299,479,799.20	+47.2%
Skipping Compressed RoaringBitmap	5,839	104,398.74	609,584,242.90	+199.7%
Compressed BitSet	5,839	111,023.99	648,269,077.60	+218.7%
Skipping BitSet	5,839	121,204.29	707,711,849.30	+247.9%
Skipping Compressed BitSet	5,839	200,746.40	1,172,158,230.00	**+476.3%**

Even though compression and skipping optimisations could be implemented for boolean lists and arrays, they are likely to be less efficient and would require significantly more memory than bit sets. A primitive boolean requires 1 byte, while a boolean object requires 16 bytes due to the object header in the JVM. In a bit set, one boolean requires only 1 bit. Accordingly, these slower and less efficient combinations have not been implemented nor tested.

Table 1 summarises the effect of each optimisation (row), measured in terms of several computational performance metrics (columns): the number of backtests per second (with generated signal blocks in the expression language), the number of bars in the input series processed per second, and the relative runtime performance (based on the bars per second) with the implementation using primitive boolean arrays as a baseline. From Table 1 we can **conclude that it is possible to improve the relative performance by up to 476.3%** when using skipping and compressing bit sets as the storage format compared to the baseline primitive boolean array. RoaringBitmap (Lemire et al., 2017) provides additional compression of sparse bit sets by not storing each boolean as a bit, but instead storing only the indexes where the bit set evaluates true. This might save memory for large and sparsely true data but comes at a performance impact of up to 48.0%. We only use it in our platform when more than 1 million data points are stored in memory because the performance penalty shrinks with these larger sizes to about 20.0%.

4.3 Expression Evaluation

In Table 2 we compare the speed of using *cached signal blocks* versus evaluating expressions during backtests. The same scenario is applied as in the above tests. Skipping and compression optimisations are not possible when the expressions are not precalculated and cached into bit sets. We also disable expression simplifications and subexpression elimination for a raw evaluation test. Such simplifications might remove unneeded calculations like "+0" or "-0" while subexpression elimination might turn constant calculations "5+3+2" into a constant value "10". Subexpression eliminations avoid repeating calculations that are already known. This should not be confused with caching or memoisation because there is no additional storage involved. Instead, it is a language feature. When bit sets are not used in favour of the evaluation of expressions, only indicators and price data are retrieved from primitive arrays in a cache that never expires. Because the cache never expires, we call the technique memoisation. Even though the blackboard engine can evict unused data (for example intermediary calculations), the heuristic is smart enough to never evict data that is actively used in our experimental setup.

Table 2: Expression evaluation performance overhead

Optimisation	**Bars**	**Backtests/s**	**Bars/s**	**Relative**
Raw (Evaluation)	5,839	14,217.35	83,015,106.65	Baseline
Subexpression Elimination and Simplification (Evaluation)	5,839	15,730.36	91,849,572.04	+10.6%
Skipping Compressed BitSet (Memoisation)	5,839	200,746.40	1,172,158,230.00	**+1,312.0%**

Table 2 shows the results of unoptimised (raw) and optimised (subexpression elimination and simplification) evaluation against memoisation (with bit sets). As in the previous table, the number of backtests was measured in a multi-threaded strategy generation process. The bars per second and the relative speed differences were calculated based on the unoptimised (raw) evaluation. We can conclude that the speed of **evaluations with memoisation and highly optimised bit sets is 1,312.0% faster or 14.1 times as fast** when evaluating the expressions for every invocation.

The raw evaluation would be significantly worse if the expression blocks included more irrelevant calculations. This would be different if mathematical or register-based expressions were generated. However, such generators have not yet been implemented in the platform and hence, are not tested here. Instead, see the performance tests in <anonymised> (2022b) for expressions where this would significantly improve the performance. A simplification that is not yet implemented but would improve generated mathematical expressions significantly would be the removal of redundant variables or functions that are neutral in the calculation. For example: "+a ...-a" or "+f() ...-f()" or "a / a" or "f() / f()".

4.4 Memory Requirements

With optimisation techniques such as caching or memoisation, there is often a trade-off between computational performance and memory efficiency. The memory consumption in bytes can be calculated as follows for our simplified boolean expression strategy generator. The calculation below is for the experimental setup of signal-based strategy generation processes that we use throughout the document. Even though the bit sets are the primary source of information for the signal-based strategies, we still have to calculate the intermediary indicators as integer/float/double/long primitive arrays. The primitive array indicators might be used directly for calculating entry levels of breakout-based strategies.

- **Primitive Array of Double or Long** (timestamp, price, indicator, exchange rate): 5,839 (bars in test) * 8 (bytes) = **46,712 (bytes)**
- **Primitive Array of Integer or Float** (count, index): 5,839 (bars in test) * 4 (bytes) = **23,356 (bytes)**
- **Bit Set of Boolean** (simple signal): 5,839 (bars in test) / 8 (1 bit) = **730 (bytes)**
- **Bars** consist of: 46,712 (start time) + 46,712 (end time) + 46,712 (first tick time) + 46,712 (last tick time) + 46,712 (open) + 46,712 (high) + 46,712 (low) + 46,712 (close) + 46,712 (volume) + 46,712 (mean) + 23,356 (mean count) = 10 * 46,712 + 1 * 23,356 = **490,476 (bytes)**
- **Ticks** consist of: 46,712 (tick time) + 46,712 (ask) + 46,712 (bid) + 46,712 (ask volume) + 46,712 (bid volume) = 5 * 46,712 = **233,560 (bytes)**
- **Indicators for Evaluation** consist of: 141 (double indicators) * 46,712 + 60 (integer indicators) * 23,356 + 119 (boolean indicators) * 739 = **8,075,693 (bytes)**
- **Signal Blocks for Memoisation** consist of: 3818 (boolean signal blocks) * 739 = **2,821,502 (bytes)**

- **Total Memory Usage:** 490,476 (bars) + 233,560 (ticks) + 8,075,693 (indicators) + 2,821,502 (signal blocks) = 8,799,729 (Blackboard Engine) + 2,821,502 (Signal Block Cache) = **11,621,231 (bytes)**

Thus we require about 12 MB of memory for a strategy generator based on about 20 years of daily data. Using primitive boolean arrays instead of bit sets we would require 19,750,514 bytes more, thus about 20 MB of additional memory or about 32 MB in total. **Bit sets save us 63.0% in memory requirements** with this strategy generator. The above calculation can be extrapolated to other time frames. Depending on the markets and their trading session times the amount of data can be significantly less than the example below which is based on a 24-hour trading session. Accordingly, it is better to directly count the given bars in a given time frame of an instrument.

- **Daily** (5,839 bars): **12 MB**
- **4 Hours** (35,034 bars): 12 MB * (24 hours / 4 hours) = 12 MB * 6 = **72 MB**
- **Hourly** (140,136 bars): 12 MB * 24 hours = **288 MB**
- **5 Minutes** (1,681,632 bars): 12 MB * 24 hours * (60 minutes / 5 minutes) = 12 MB * 24 * 12 = **3,456 MB (3.5 GB)**
- **1 Minute** (8,408,160 bars): 12 MB * 24 hours * 60 minutes = **17,280 MB (17.3 GB)**

Nowadays such memory requirements can be easily fulfilled with even laptops having 128 GB of available memory (Schenker Technologies, 2022). Reasonably priced servers also provide this amount of memory (HetznerOnlineGmbH, 2022). There is also space for testing multiple instruments together in baskets or using more indicators and caching more signal blocks. During walk-forwards, one also does not need to load the entire data set into memory. Instead, each segment can be loaded individually to, for example, test on tick data and generate strategies each month with a strategy generator that uses the last 3 months of available ticks. Between each segment, the previous data is unloaded to make space for the next in-memory segment. Also, multiple instruments can be optimised separately so that only one instrument is loaded into memory at a time. However, with an investment into more memory, segments could be kept in memory between runs to speed up multiple runs for robustness tests. This is easily achievable for more common data resolutions which are still considered intraday trading. The JVM still needs memory to work with to not be slowed down significantly due to garbage collection. As a rule of thumb, the maximum heap size should be allocated at about double the calculated residual memory requirement of the data set.

The CPU cache will likely not be able to hold all of the data, even that of a daily frequency. The column-oriented primitive array storage is optimised to maximise the use of CPU prefetching so that data is preloaded from RAM before the calculation requires it. In theory, it would be possible to load larger data sets into virtual memory than in the hardware memory (RAM) available on a computer. However, swapping to disk will likely cause backtest speeds to drop below speeds that are possible with classical backtesting engines that do not load all the data into memory. This is because the operating systems might not be intelligent enough about what memory to prioritize, which results in memory thrashing. A way around this could be to make use of memory compression which promises up to 40% less memory requirements while maintaining 40% to 70% of the system performance (Jennings, 2013). However, it might be cheaper to stay within the limits of the hardware (by chunking backtests) or to increase the hardware memory instead of paying (e.g. in cloud computing instances billed by minutes) for longer backtest times due to degraded performance. This is why we have not yet verified or tested these workarounds for larger data sets.

5 Performance Results

In this section, we compare the runtime performance of our Invesdwin platform with competitor platforms. For head-to-head performance results, we use *event-based strategy APIs* provided by the respective platforms, rather than using our expression language, to have a fair comparison with other platforms that do not have

specialised expression languages. Our expression language is built as a more accessible simplification on top of our event-based strategy API, providing the same functionality. For the measurements in this section, the strategies are coded by hand in a feature-rich programming language instead of having them generated by an expression language. We focus on the raw performance of classical backtesting engines and their optimisation steps, specifically the difference between storage formats, the potential for multi-threading, and the effect of changing data resolutions in strategy optimisation tasks. Later in Section 5.4 we focus on the performance in the context of a full ML backtesting engine, and there we will revisit our expression language.

Each test is executed after a few warmup runs of backtests in the given platform to load data, populate caches, and perform pre-calculations. We repeat the test 10 times and record the best time. Platforms are included in the comparison only if a free demo or trial version is available. Some platforms were omitted because it was not possible to do reliable tests for processing tick data in that given scenario, or because a licensed version for the purpose was not readily available. The data sources of the given platforms differ in the number of ticks (discrete data points) available. Accordingly, a backtest duration was chosen to provide a sufficiently large sample of ticks. The same data was imported where possible. Statistics gathering was minimised. The main *metric* for our evaluation is the *achieved rate of Ticks per Second (Ticks/s).*

5.1 High Data Resolution: Ticks

Table 3: **Raw performance** in bars or ticks per second (ticks/s)

Platform	**Bars or Ticks**	**Seconds**	**Bars/s or Ticks/s**	**Relative**
Daily Data:				
Python: Zipline-Reloaded	4,528	5.098	888.19	-84.5%
Python: Backtrader2	4,528	0.789	5,738.91	*Baseline*
Python: PyAlgoTrade	4,528	0.145	31,117.59	+442.2%
Minutes Data:				
Python: Backtrader2	1,000,000	62.020	16,123.83	-93.0%
JForex 4	1,052,540	46.581	22,598.05	-90.1%
Python: PyAlgoTrade	2,000,000	68.263	29,298.45	-87.2%
MetaTrader 4	1,117,631	6.484	172,367.52	-27.8%
Zorro S	1,088,716	4.750	229,203.37	*Baseline*
MetaTrader 5	1,117,231	0.494	2,261,601.21	+886.7%
Invesdwin (Historical)	1,486,140	0.341	4,358,181.82	+1,701.4%
Invesdwin (Blackboard)	1,486,140	0.149	9,974,093.96	**+4,251.6%**
Ticks Data:				
JForex 4	16,797,607	93.778	179,120.98	-91.9%
Zorro S	17,479,849	7.920	2,207,051.64	*Baseline*
MetaTrader 5	16,797,607	3.098	5,422,081.02	+145.7%
Invesdwin (Historical)	24,232,002	3.771	6,425,882.26	+191.1%
Invesdwin (Blackboard)	24,232,002	1.727	14,031,268.67	**+535.7%**

Table 3 shows performance results from the event-based API backtest, processing about one year's worth of ticks (discrete data points) on EUR/USD exchange rates *without* executing trades or strategy logic. This measurement focuses on the *raw performance of processing data points* without capturing the overhead of a transaction or the complexity of the strategy logic. Because not all contender platforms support ticks as the primary data feed, we also compare them with 1-minute bar feeds (aggregated data) with 3 years' worth of bars.

We could not get Zipline-Reloaded to run backtests on 1-minute data due to session alignment issues. Instead, we used a daily feed and compared that against the other Python-based platforms. These numbers indicate that most purely Python-based platforms, without tuned, underlying libraries, are too slow to be competitive in our comparisons.

With MetaTrader 4 we were unable to ensure that "processing all ticks" actually loaded real ticks instead of interpolating artificial ticks from bars, thus we don't classify this as testing as ticks. Zorro S provides a similar interpolation mode, which we exclude from our tests as well in favour of real ticks and 1-minute bar tests that can be configured in Zorro S.

For the relative comparison, we use *Zorro S as the baseline* because it also loads the data fully into memory similar to our blackboard engine. The results in Table 3 show that our **blackboard engine is 535.7% faster or 6.3 times as fast for simply processing ticks**.

The data in Table 4 and its visualisation as bar-charts in Figure 8 show the event-based API backtest engines executing trades in ticks resolution from a simple strategy (*Moving Average Crossover*) performing the logic of the strategy on 1-minute bar intervals. Again, we include tests in 1-minute data feeds because not all contender platforms support testing on ticks. We also added vector-based backtesting engines for comparison here. It makes no sense to test vector-based engines on raw data processing speeds as above since calculations are required to make a pass through the data. That is why we test them using this simple strategy instead. The tabulated results show that our **blackboard engine is 1,303.6% faster or 14.0 times as fast compared to Zorro S for backtesting a simple strategy on ticks**.

Table 4: Performance of **Moving Average Crossover** strategy single backtest (bars or ticks)

Platform	**Bars or Ticks**	**Seconds**	**Bars/s or Ticks/s**	**Relative**
Daily Data:				
a) Python: Zipline-Reloaded	4,528	16.413	275.88	-93.2%
b) *Python: Backtrader2*	4,528	1.114	4,064.63	*Baseline*
c) Python: PyAlgoTrade	4,528	0.165	27,442.42	+575.1%
d) Python: vectorbt	4,528	0.039	116,104.56	+2,756.4%
Minutes Data:				
e) R: quantmod	200,000	20.248	9,877.52	-91.7%
f) Python: Backtrader2	1,000,000	94.977	10,528.86	-91.2%
g) JForex 4 (Minutes)	1,052,540	50.696	20,763.77	-82.7%
h) Python: PyAlgoTrade	2,000,000	86.709	23,065.66	-80.7%
i) TradeStation 9.5	6,925,170	290.880	23,807.65	-80.1%
j) Python: vectorbt	2,000,000	17.257	115,895.00	-3.2%
k) *Zorro S*	1,088,716	9.090	119,770.73	*Baseline*
l) MetaTrader 4	1,117,631	6.969	160,371.79	+33.9%
m) Matlab	3,002,266	10.917	275,008.34	+129.6%
n) NinjaTrader 8	370,801	1.240	299,033.06	+149.7%
o) Julia: Strategems.jl	3,002,266	6.475	463,670.42	+287.1%
p) MetaTrader 5	1,117,231	2.239	498,986.60	+316.6%
q) Invesdwin (Historical)	1,486,140	1.763	842,960.86	+603.8%
r) Invesdwin (Blackboard)	1,486,140	0.511	2,908,297.46	**+2,328.2%**
Ticks Data:				
s) JForex 4	16,797,607	102.586	163,741.71	-74.9%
t) *Zorro S*	17,479,849	26.780	652,720.28	*Baseline*
u) FXCM Trading Station	2,976,026	4.080	729,418.13	+11.7%
v) NinjaTrader 8	16,472,101	8.190	2,011,245.54	+208.3%
w) MetaTrader 5	16,797,607	7.567	2,219,850.27	+240.1%
x) Invesdwin (Historical)	24,232,002	6.866	3,529,274.98	+440.7%
y) Invesdwin (Blackboard)	24,232,002	2.645	9,161,437.43	**+1,303.6%**

In general, we observe that the Python- and R-based platforms show the poorest performance. Customised platforms such as Tradestation, MetaTrader, and Zorro S consistently outperform these platforms. Our platform achieves considerable gains on top of these platforms on both minutes and tick data, and our optimisations summarised in Section 4 are instrumental in achieving this performance.

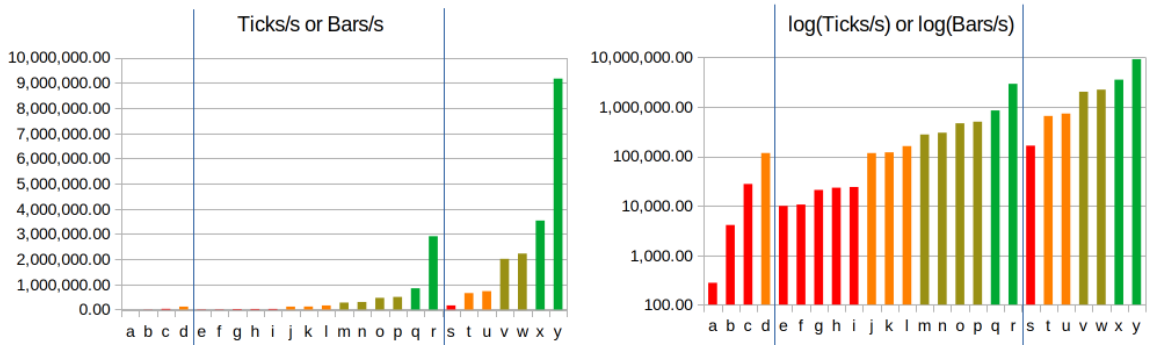


Figure 8: Graphical representation (linear and log y-scale) of Table 4: Performance of **Moving Average Crossover** strategy single backtest (bars or ticks); bars q, r, x and y represent our Invesdwin platform

Zorro S is an interesting platform because it preloads all data into memory before executing a backtest. While speeding up the backtesting, this also limits the window for backtesting because at the time of writing Zorro was only available as a 32-bit application. Our tests were performed with an older licensed version of Zorro S (1.83), where we limited the data to a size that fits into the backtesting engine of Zorro.[8] Zorro has many backtesting features in common with our Invesdwin platform and it allows fine-grained control over how data feeds are processed in backtests. Our platform is written in Java and thus fully supports 64-bit memory allocations. We are only limited by the available (virtual) memory with our blackboard engine that also loads all data into memory. With the historical engine, any amount of data can be tested because it loads data in a moving window from data files.

As a more substantial case study, we have also implemented the *Workshop 5* strategy from the Zorro documentation (oPgroupGermanyGmbH, 2022a) to compare backtesting speeds in optimisation scenarios. The strategy in Workshop 5 is a *moving averages crossover* strategy that uses two indicators with a maximum lookback of 500 bars for the normalisation of a smoothing filter algorithm. It enters trades based on a *crossover* with a threshold and uses a stop loss based on a volatility indicator. The execution of the trailing stop loss is disabled for these tests on both platforms. The same indicators, algorithms and settings are used in both cases. Regarding parallelization, Zorro can not utilize multiple threads for optimisation backtests. This is only possible in walk-forward analysis runs. Zorro is not the only platform that does not fully utilize multiple threads in that scenario. In contrast, our platform supports multi-threading in more testing scenarios than other platforms. To have a fair comparison, we compare both platforms in single-threaded execution but also state numbers for the Invesdwin platform when multiple threads are used. The first backtests are performed with ticks precision, while the strategy executes trades and strategy logic in four-hour bar intervals. Full statistics are collected in memory in these backtests, though report creation is excluded from the measured time of both platforms. Our platform collects significantly more statistics than Zorro S and that is part of the measured time.

First, we compare the single backtest scenario in Table 5. We use the EUR/USD exchange rates in the year 2015 as the source of the ticks. The results show that our **blackboard engine is 421.3% faster or 5.2 times as fast compared to Zorro S for backtesting the *Workshop 5* strategy on ticks**.

The *Precalculated* optimisation uses file storage for indicator results and retrieves these values from the file on successive backtests instead of calculating the indicators individually in each run. Storing the data on a hard disk or SSD does not make much of a difference because the operating system file cache keeps hot segments in memory. Also, the series only consists of a long timestamp and a double value over the whole time range at four-hour intervals (the resolution of strategy decisions). So the data will be small enough to fit

[8] A 64-bit version of Zorro S (2.50) has become available in the meantime (which our existing license does not include) that supports bigger data sets. The changelog notes slightly improved backtesting speeds due to better memory management (oPgroupGermanyGmbH, 2022b). However, we expect this effect to not change the results by a magnitude.

Table 5: Performance of Workshop 5 **single backtest** (ticks)

Platform	**Ticks**	**Seconds**	**Ticks/s**	**Relative**
Zorro S	34,443,682	53.050	649,268.27	*Baseline*
Invesdwin (Historical)	24,232,002	41.946	577,695.18	-11.0%
Invesdwin (Historical, Precalculated)	24,232,002	30.605	791,766.12	+21.9%
Invesdwin (Blackboard)	24,232,002	7.159	3,384,830.56	**+421.3%**

into memory after the first run with a size of about 0.5 MB.[9] The most benefit comes from not requiring to calculate complex indicators multiple times. This effect becomes more significant when more backtests can share these precalculations. This optimisation is not needed for the blackboard engine because everything is loaded from precalculated primitive arrays in memory with that engine.

In Table 6, we compare a small optimisation run where three parameters specific to the strategy (smoothing filter bars, crossover threshold, volatility multiplier for the stop loss) are optimised in 10% increments for the values (though the platforms calculate this differently). This is an example of important parameter tuning as needed for an ML learning platform. Each parameter is optimised individually which results in 32 backtests in our engine. Zorro executes each backtest sequentially, while the Invesdwin platform can run multiple backtests in the same thread simultaneously. This allows for the re-use of the same data loading pipelines over multiple backtests. In this way, **reducing file access** achieves higher throughput than running each backtest individually. When multiple cores are available Invesdwin will chunk backtests between the available threads. Multiple strategies then run on the same data stream covering the whole time range in one thread. This allows a higher throughput while each thread can reuse data pipelines for its backtests that run together. For the blackboard engine, this optimisation is disabled because in memory it is faster to execute each backtest separately due to improved usage of CPU prefetching since the CPU caches are limited in size. So the blackboard engine still makes use of multi-threading but only runs a single strategy per thread. Regardless of how many strategies run together in a thread, when a thread finishes its work, it will continue with the next strategy or chunk of strategies until all required backtests are done. Since a file buffer cache has been introduced in our NoSQL database it is faster to only run a few (e.g. 5) backtests together per thread in the historical engine, because the overhead of accessing the files is already reduced considerably by the database. The results show that our **blackboard engine is 1,731.0% faster or 18.3 times as fast compared to Zorro S for step-wise optimising the *Workshop 5* strategy on ticks**.

Table 6: Performance of Workshop 5 **step-wise optimisation** (ticks)

Platform	**Backtests**	**Seconds**	**Backtests/s**	**Ticks/s**	**Relative**
Zorro S	38	798.210	0.048	1,639,743.82	*Baseline*
Invesdwin, 1 Thread:					
Historical	32	572.197	0.056	1,355,169.75	-17.3%
Historical, Precalculated	32	427.648	0.075	1,813,229.72	+10.5%
Blackboard	32	155.639	0.206	4,982,196.39	+203.8%
Invesdwin, 12 Threads:					
Historical	32	112.054	0.285	6,920,092.67	+322.0%
Historical, Precalculated	32	91.129	0.351	8,509,081.24	+418.9%
Blackboard	32	25.382	1.261	30,550,156.17	**+1,731.0%**

Table 7 shows how the Invesdwin platform *scales* when optimising the three parameters together in a brute force fashion, i.e. exploring all possible combinations of parameter values. This multiplies the permutations and consequently requires significantly more backtests (1331 in our case). We stick to multi-threaded execution for these tests. The platform has heuristics to allow only a specific number of parallel backtests so as not to exhaust the available memory. If there are more backtests scheduled than possible to run simultaneously,

[9] If it does not fit for decisions made in higher data resolutions (e.g. ticks), it will still improve results if the calculations that are skipped are costly enough to offset the disk input/output overhead.

the backtests will run in multiple step-wise chunks. For the relative comparison, we use the same baseline that we used in the above Table 6 because the backtesting speed does not change significantly when more backtests are to be run in Zorro S. The results show that our **blackboard engine is 1,285.6% faster or 13.8 times as fast compared to Zorro S for brute force optimising the *Workshop 5* strategy on ticks**. In practical terms, we only have to wait about 24 minutes, rather than more than 7 hours, for 1331 backtests.

Table 7: Performance of Workshop 5 **brute force optimisation** (ticks)

Platform	**Backtests**	**Seconds**	**Backtests/s**	**Ticks/s**	**Relative**
Invesdwin, 12 Threads:					
Historical	1331	5,478.488	0.242	5,887,170.81	+259.0%
Historical, Precalculated	1331	3,133.338	0.425	10,239,429.77	+524.4%
Blackboard	1331	1,419.571	0.937	22,720,099.71	**+1,285.6%**

5.2 Medium Data Resolution: One Minute Bars

Normally one has to balance a high data resolution for trustworthy backtest results against the performance overhead of processing so many data points. Not all platforms support algorithmic trading strategies on ticks or are limited by how many ticks can be processed in a single backtest. Often the compromise is to run tests on 1-minute bars. We thus test the previous scenario against the Zorro platform by running both optimisations on 1-minute bars on EUR/USD exchange rates, this time from 2003 to 2015 to get enough sample bars. First, we compare single runs in Table 8. The results show that our **blackboard engine is 7,112.4% faster or 72.1 times as fast compared to Zorro S for backtesting the *Workshop 5* strategy on 1-minute bars**.

Table 8: Performance of Workshop 5 **single backtest** (one minute bars)

Platform	**Bars**	**Seconds**	**Bars/s**	**Relative**
Zorro S	6.177.906	152.700	40,457.80	*Baseline*
Historical	4,747,531	12.738	372,706.15	+821.2%
Historical, Precalculated	4,747,531	10.319	460,076.65	+1,037.2%
Blackboard	4,747,531	1.627	2,917,966.19	**+7,112.4%**

Second, we run the step-wise optimisation of three parameters (individually after each other) in Table 9. The Invesdwin tests are directly performed with 12 threads. The results show that our **blackboard engine is 14,364.3% faster or 144.6 times as fast compared to Zorro S for step-wise optimising the *Workshop 5* strategy on 1-minute bars**.

Table 9: Performance of Workshop 5 **step-wise optimisation** (one minute bars)

Platform	**Backtests**	**Seconds**	**Backtests/s**	**Bars/s**	**Relative**
Zorro S	38	3,719.270	0.010	63,120.03	*Baseline*
Invesdwin, 12 Threads:					
Historical	32	59.123	0.541	2,569,575.16	+3,970.9%
Historical, Precalculated	32	51.136	0.626	2,970,920.53	+4,606.8%
Blackboard	32	16.640	1.923	9,129,867.31	**+14,364.3%**

We also repeat the same brute force optimisation of all three parameters together (multiplied permutations) in Table 10. The results show that our **blackboard engine is 16,001.9% faster or 161 times as fast compared to Zorro S for brute force optimising the *Workshop 5* strategy on 1-minute bars**.

Table 10: Performance of Workshop 5 **brute force optimisation** (one minute bars)

Platform	Backtests	Seconds	Backtests/s	Bars/s	Relative
Invesdwin, 12 Threads:					
Historical	1331	2,027.233	0.656	3,117,038.72	+4,838.3%
Historical, Precalculated	1331	1,421.373	0.936	4,445,675.95	+6,943.2%
Blackboard	1331	621.730	2.141	10,163,517.54	**+16,001.9%**

5.3 Low (Minimum) Data Resolution: Four Hour Bars

Since the strategy makes decisions in four-hour intervals, we can also test by using only bars in that interval. This is the most efficient data resolution to use as it reduces the number of data points, collapsing them into intervals. The strategy makes the same decisions as in higher data resolutions but works with a data set on bars (intervals) rather than ticks (points). The backtest results will be the same for our platform regardless of the data resolution chosen as long as the minimum for the strategy logic is not exceeded. Other platforms might produce varying or even wrong results in lower data resolutions. We don't have this issue because we use the historical spread from ticks when executing trades based on bars. This is done by remembering the related ticks for each bar and using them during trade execution even in lower data resolution tests. We call this feature "Skipping Ticks".[10] It increases the work in our backtesting engine due to loading ticks and bars as separate data streams. This feature can be turned off to behave like other platforms with reduced accuracy and make our backtests up to 15% faster (when compared against our backtests with "Skipping Ticks" here, not against Zorro S).

First, we compare single runs in Table 11. Again we use EUR/USD from 2003 to 2015 to get a large enough sample of bars. Our **blackboard engine is 3,534.6% faster or 36.3 times as fast compared to Zorro S for backtesting the *Workshop 5* strategy on four-hour bars**.

Table 11: Performance of Workshop 5 **single backtest** (four hour bars)

Platform	Bars	Sec	Bars/s	Relative
Zorro S	17,652	1.640	10,763.41	*Baseline*
Historical	21,125	0.409	51,650.37	+479.9%
Hist, Precalc	21,125	0.152	138,980.26	+1,191.2%
Blackboard	21,125	0.054	391,203.70	**+3,534.6%**

Second, we run the step-wise optimisation of 3 parameters in Table 12. The Invesdwin tests are directly performed with *12 threads.* The results show that our **blackboard engine is 14,745.1% faster or 148.4 times as fast compared to Zorro S for step-wise optimising the *Workshop 5* strategy on four-hour bars**.

Table 12: Performance of Workshop 5 **step-wise optimisation** (four hour bars)

Platform	Backtests	Seconds	Backtests/s	Bars/s	Relative
Zorro S	38	45.520	1.197	14,735.85	*Baseline*
Invesdwin, 12 Threads:					
Historical	32	2.951	10.844	229,074.89	+1,454.5%
Historical, Precalculated	32	0.762	41.995	887,139.11	+5,920.3%
Blackboard	32	0.309	103.560	2,187,702.26	**+14,745.1%**

[10] A trailing stop loss that uses every tick as the minimum data resolution will have varying results on coarser bars even in our platform. Using a trailing stop that only updates every four hours would make results consistent even if finer resolutions were to be used. So it depends on the strategy design, particularly on when decisions are allowed to be made.

We also repeat the same brute force optimisation of all 3 parameters together in Table 13. The results show that in this configuration our **blackboard engine is 17,319.1% faster or 174.2 times as fast compared to Zorro S for brute force optimising the *Workshop 5* strategy on four-hour bars**.

Table 13: Workshop 5 **brute force optimisation** (four hour bars)

Platform	**Backtests**	**Seconds**	**Backtests/s**	**Bars/s**	**Relative**
Invesdwin, 12 Threads:					
Historical	1331	226.607	5.874	124,079.90	+742.0%
Historical, Precalculated	1331	22.423	59.359	1,253,952.41	+8,409.5%
Blackboard	1331	10.954	121.508	2,566,859.14	**+17,319.1%**

5.4 ML Performance using Genetic Programming

After having examined the performance characteristics of classical backtesting engines, we can now demonstrate how the performance compares to specialised machine learning backtesting engines. In this section, we switch from classical strategy development to AI-generated strategies using genetic programming. A simplistic view is that faster backtests find better strategies, but this is not always the case. Faster backtesting allows one to faster find suitable candidate strategies from the problem space given by the strategy generator. Looking at more candidate strategies might lead to a higher risk of curve fitting and selection bias. This could in turn to an increase in the false negative (Type II error) rate where bad strategy candidates are not rejected often enough. Thus, speed is only one part of the strategy generation process. The other important part is *robustness testing* and portfolio selection, which should mitigate these biases in the process. Our platform's mid term goal is to research these topics, but a detailed analysis is out of the scope for this paper. With this caveat, we focus here on our improvements to the backtesting speed, which enables us to research the robustness of strategy development processes. Faster backtesting speed also allows us to spend more computational time on extensive robustness tests.

The data in Table 14 and its visualisation as bar-charts in Figure 9 compares the Invesdwin machine learning backtesting engine with other platforms that have this capability. Since all platforms can essentially generate candidates endlessly, it makes no sense to capture the total run time. Instead, we let the platforms run for a while to generate a large enough sample and record the performance. We measure the *backtests per second* as a measure of performance and derive from that the processed bars per second. The values are subsequently rounded. This means that not all processed bars might be touched by the platform since specific optimisations can skip chunks of data and arrive at the same result for strategy backtests. However, this kind of comparison allows us to compare normalised performance as bars per second regardless of such optimisations (that differ across platforms). We configure 100% in-sample periods so that the genetic programming algorithm uses all bars. We use varying numbers of bars to assess the overhead of genetic programming. Precalculation and warmup are not counted. All platforms use multiple threads and can utilize the CPU fully. BuildAlpha and Invesdwin generate signal-based strategies with 4 entry and 4 exit blocks combined with identical settings. StrategyQuantX generates signal-based strategies with 4 entry blocks and a time-based exit. Adaptrade and GeneticSystemBuilder generate mathematical entry expressions with 4 indicators compared against a threshold with a time-based exit.

The population size and other genetic programming parameters are left at their default in the respective platform because this should not influence the final number of bars per second. It does not matter in what form the backtests are performed, as long as the threads all mainly execute backtests. We don't measure the number of candidate strategies created. Instead, we measure how many backtests were executed. Also, the total number of backtests executed is irrelevant because we just collect a large enough sample (a few minutes) to get the number of backtests executed per second.

For the relative comparison, we use GeneticSystemBuilder (intraday) as the baseline because it has a similar performance to the test of the classical backtesting engines in Section 5. Our machine learning engine is based on the blackboard version of our classical backtesting engines from Section 5. The data in Table 14 and Figure 9 shows that **Invesdwin (intraday) is 59,503.8% faster or 596.0 times as fast compared**

to GeneticSystemBuilder (intraday). And **Invesdwin (20 years) is 554.8% faster or 6.5 times as fast compared to BuildAlpha (20 years)**.

Table 14: ML performance: strategy generator performance (12 threads)

Platform	Bars	Backtests/s	Bars/s	Relative
a) GeneticSystemBuilder (4 years)	1,296	113,82	147,507.40	-94.0%
b) Adaptrade (4 years)	1,296	215.61	304,654.44	-87.5%
c) StrategyQuantX (20 years)	6,164	50.33	310,266.73	-87.3%
d) Adaptrade (30 years)	10,336	42.16	435,750.42	-82.2%
e) StrategyQuantX (intraday)	6,272,874	0.08	472,923.58	-80.6%
f) GeneticSystemBuilder (30 years)	10,336	77.70	803,108.00	-67.1%
g) *GeneticSystemBuilder (intraday)*	3,222,007	0.76	2,443,505.99	*Baseline*
h) BuildAlpha (1 year)	259	143,530.00	37,174,270.00	+1,421.3%
i) BuildAlpha (4 years)	1,098	104,920.00	115,202,160.00	+4,614.6%
j) BuildAlpha (20 years)	6,284	27,420.00	172,307,280.00	+6,951.6%
k) Invesdwin (1 year)	314	916,669.56	287,834,241.80	+11,679.6%
l) Invesdwin (4 years)	1,252	596,633.89	746,985,630.30	+30,470.2%
m) Invesdwin (20 years)	5,839	193,252.58	1,128,401,815.00	+46,079.6%
n) Invesdwin (intraday)	1,482,424	982.46	1,456,422,283.00	**+59,503.8%**

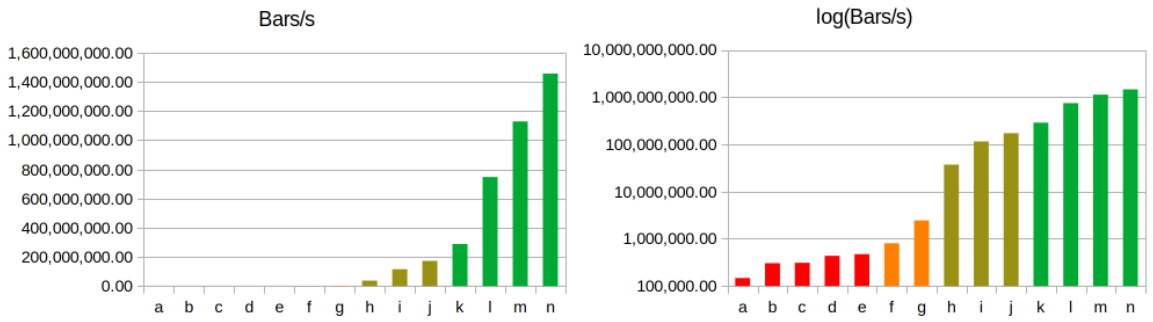


Figure 9: Graphical representation (linear and log y-scale) of Table 14: Strategy generator performance (12 threads); bars k–n represent our Invesdwin platform

BuildAlpha (bars h–j) was an inspiration in the design of the Invesdwin platform machine learning backtesting engine because it showed that significantly higher backtest speeds are possible with a specialised backtesting engine for machine learning. Other platforms seemingly use classical event-based backtesting engines judging by the measured speeds. Invesdwin makes use of more advanced optimisations that increase the speed further than BuildAlpha, especially *bit set compression and skipping of false indexes.* Both platforms use simplified boolean expressions as the basis for the strategy generator.

In the Invesdwin platform, it is also possible to implement more powerful strategy generators like the mathematical threshold indicator calculation that GeneticSystemBuilder uses or the breakout strategies that StrategyQuantX can generate. These generators will still run magnitudes faster in Invesdwin even though not all optimisations can be fully utilised. This is because other platforms implement only a subset of possible optimisations. Both StrategyQuantX and Invesdwin generate breakout strategies with 4 signal filters and a time-based exit in the tests reported below. Table 15 shows that **Invesdwin is 245,399.6% faster or 2,455 times as fast compared to StrategyQuanX for the more sophisticated breakout strategies**.

5.5 Main Contributors to Performance Improvement

To separate the main contributors to the performance improvements, we can pick important measurements of the previous sections and tabulate them for relative improvements. Table 16 and Figure 10 show these

Table 15: ML performance: breakout generator performance

Platform	Bars	Backtests/s	Bars/s	Relative
StrategyQuantX (Breakout)	6,164	42.35	261,073.17	Baseline
Invesdwin (Breakout)	5,839	109,767.72	640,933,717.10	**+245,399.6%**

improvements measured between each step in relative terms. This is not a comparison based on the same problem, but it allows us to visualise our journey to achieving the backtesting speeds we currently have. Changing from a classical backtesting engine (historical) to an in-memory backtesting engine (blackboard) improved backtesting speeds by a factor of 2.74. Applying multi-threading (12 threads) improved the speed by a factor of 6.13. Switching to our custom expression language allowed us to gain an additional factor of 2.71. Optimising these expressions only gained us a factor of 1.10 in terms of backtests, though switching to bit sets for the memoisation of calculations improved the speed by a factor of 2.18. Compressing these bit sets gained an additional factor of 3.23 and coupling that with our skipping heuristic led to the final factor of 1.80. The total of all these improvements leads us to a **gain of 646.45 times** in terms of backtesting speed. Taking our performance measurements from the JForex platform as a starting point, we **actually gained about 6543.95 times** in terms of speed.

Table 16: Relative performance contributions

Platform	Bars/s or Ticks/s	Relative
JForex:		
a) 1 Thread (Ticks)	179,120.98	-90.1%
Historical Engine:		
b) 1 Thread (Ticks)	1,813,229.72	Origin
Blackboard Engine:		
c) 1 Thread (Ticks)	4,982,196.39	+174.5%
d) 12 Threads (Ticks)	30,550,156.17	+513.2%
Expression Engine:		
e) Raw Evaluation (20 years)	83,015,106.65	+171.7%
f) Optimised Evaluation (20 years)	91,849,572.04	+10.6%
g) BitSet (20 years)	200,673,409.00	+118.5%
h) Compressed BitSet (20 years)	648,269,077.60	+223.0%
i) Skipping Compressed BitSet (20 years)	1,172,158,230.00	+80.8%

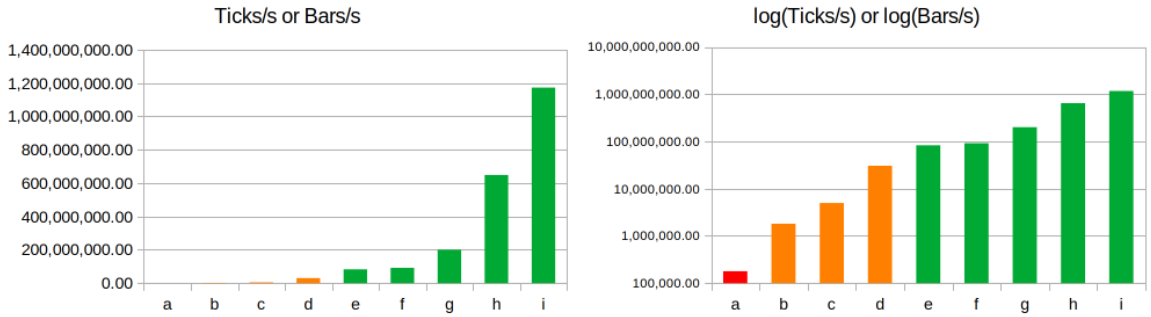


Figure 10: Graphical representation (linear and log y-scale) of Table 16: Relative performance contributions

6 Conclusions

To tackle the FinTech challenge of testing a huge number of automatically generated strategies in a machine learning framework, we developed and measured a compute engine for high-performance machine learning within our Invesdwin platform.

Faster backtesting engines allow us to increase the automation of strategy development processes and conduct research at a higher level of abstraction, tackling concrete questions in economics. We discussed in detail the design of the Invesdwin platform and how machine learning concepts can be tested with this underlying compute engine. As a core component of the platform, an expression language has been designed with which trading strategies can be generated and portfolio decisions automated. Hybrid machine learning approaches can be defined easily and tested quickly with this approach. Performance considerations and key contributors to improving the speed of empirical research have been highlighted. Specifically using **skipping compressed bit sets improves the strategy generation speed by a factor of 5.7 and saves 63% of memory** for the signal generator. This approach is 14.1 times as fast compared to raw expression evaluation (see Table 2). All of our tests were conducted on a single laptop, though the platform has yet untapped potential for parallelism (beyond multi-threading) with grid computing and cloud scaling. This is facilitated by low latency and high throughput native communication channels at the core of the compute engine.

Some of these optimisations are novel to the field of testing expression-based strategies. Specifically, the in-memory machine learning engine for a new high-performance expression language. Key concepts are the usage of a compressed "bit set" format for storage and the ability to skip parts of the storage based on data set and strategy. These compact bit sets can be loaded quickly from financial databases. We use a custom-developed, compressed NoSQL database for efficient local data storage. We demonstrate computational performance benefits by comparing existing contender platforms against our novel solution. Our in-memory (blackboard) engine for classical event-based trading strategies is faster than any of the platforms we could make available for testing. In an in-depth comparison with **Zorro S** (which has similar features in comparison to our platform), we can process ticks 3.0 (*Zorro Workshop 5*) to 14.0 (*Moving Average Crossover*) times as fast in a single thread depending on the tested strategy. **We can optimise strategies step-wise 18.3 times as fast on ticks, 144.6 times as fast on 1-minute bars, and 148.4 times as fast on four-hour bars** (measured on *Zorro Workshop 5*). This is achieved without any qualitative difference in the execution of the trades regardless of the data resolution. Thus, we can use the coarsest granularity of bars supported by a strategy if desired. With that, we can expect similar results to execution on ticks while having a significantly faster backtesting speed.

Other platforms that can generate trading strategies also use classical event-based backtesting engines. Our platform shows even larger performance gains in these benchmarks: the "Invesdwin (intraday)" test is **596.0 times as fast compared to "GeneticSystemBuilder (intraday)"**. For signal strategies, our "Invesdwin (20 years)" test is **2,589.6 times as fast compared to "Adaptrade (30 years)"**, and **3,636.9 times as fast compared to "StrategyQuantX (20 years)"**. For breakout strategies, our "Invesdwin (20 years)" test is 2,455.0 times as fast compared to "StrategyQuantX (20 years)". This is strong evidence that this level of high performance has not been previously realised in a compute engine for FinTech. Even the fastest available contender "BuildAlpha", which also uses a specialised signal strategy generator, does not use advanced optimisations like skipping compressed bit sets. This is the main reason why our "Invesdwin (20 years)" test is **6.5 times as fast compared to "BuildAlpha (20 years)"**, the fastest of the contender platforms. Beyond measuring raw speed, we observe that all contender platforms cannot formally automate and test the robustness of strategy development processes. Based on these comparisons, we have achieved the fastest backtesting performance and the highest degree of automation for researching strategy development processes.

Future Work

Additional research to investigate strategy development processes from an economics point of view is underway. Risk management, position sizing, and equity curve trading are decision points that can also be automated with specialised expressions soon. This allows for research into the design and assessment of novel hybrid ML techniques and the construction of robust investment portfolios. This research will extend the

platform with further algorithms and testing capabilities. The authors anticipate these capabilities will find significant interest in the academic and practitioner communities. The aspiration is for this novel platform to become a widely-used tool in academic research and professional practice that improves the ability of market participants to investigate new approaches in portfolio management through the adaptation of more data-driven methodologies.

Tool and Data Availability

Interested researchers can get free access to the platform (on github.com) for research purposes by emailing "<anonymised>". The data used in this paper can be downloaded manually at DukascopyBankSA (2022a) or accessed through an API with a free demo account at DukascopyBankSA (2022b).

7 References

Fatai Anifowose. Hybrid machine learning explained in nontechnical terms, 2020. URL `https://jpt.spe.org/hybrid-machine-learning-explained-nontechnical-terms`. Retrieved at 08.04.2022.

Fatai Adesina Anifowose, Jane Labadin, and Abdulazeez Abdulraheem. Hybrid intelligent systems in petroleum reservoir characterization and modeling: the journey so far and the challenges ahead. *Journal of Petroleum Exploration and Production Technology*, 7(1):251–263, 2017.

<anonymised>. Historical Cache for sparse time series with gaps in the Invesdwin Platform, 2022a. URL `https://github.com/invesdwin/invesdwin-util#caches`. Retrieved at 25.02.2022.

<anonymised>. A Simple Expression Language for Automating Strategy Development Processes. 03 2022b. doi: 10.13140/RG.2.2.18081.48484. URL `https://www.researchgate.net/publication/359659846_A_Simple_Expression_Language_for_Automating_Strategy_Development_Processes`. Retrieved at 01.04.2022.

<anonymised>. Synchronous Channels in the Invesdwin Platform, 2022c. URL `https://github.com/invesdwin/invesdwin-context-integration#synchronous-channels`. Retrieved at 20.02.2022.

<anonymised>. Specialised NoSQL Database for financial time series in the Invesdwin Platform, 2022d. URL `https://github.com/invesdwin/invesdwin-context-persistence#timeseries-module`. Retrieved at 20.02.2022.

Rob Arnott, Campbell R. Harvey, Vitali Kalesnik, and Juhani Linnainmaa. Alice's Adventures in Factorland: Three Blunders That Plague Factor Investing. *The Journal of Portfolio Management*, 45(4):18–36, 2019.

D. Aronson. *Evidence-Based Technical Analysis: Applying the Scientific Method and Statistical Inference to Trading Signals*. Wiley Trading. Wiley, 2011. ISBN 9781118160589.

David Aronson and Timothy Masters. *Statistically Sound Machine Learning for Algorithmic Trading of Financial Instruments: Developing Predictive-Model-Based Trading Systems Using TSSB*. CreateSpace Independent Publishing Platform, ISBN: 978-1489507716, 2013. ISBN 148950771X.

Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLoS ONE*, 12, 07 2017. doi: 10.1371/journal.pone.0180944.

Dave Bergstrom. BuildAlpha, 2020. URL `https://www.buildalpha.com`. Retrieved at 17.04.2020.

Jonathan Blackledge, Kieren Murphy, Currency Traders Ireland, and Dublin Docklands Innovation Park. Forex trading using metatrader 4 with the fractal market hypothesis. In *Proceedings of the Third International Conference on Resource Intensive Applications and Services, INTENSIVE*, pp. 1–9, 2011.

Indranil Bose and Radha K. Mahapatra. Business data mining — a machine learning perspective. *Information & Management*, 39(3):211–225, 2001. ISSN 0378-7206. doi: 10.1016/s0378-7206(01)00091-x. URL `https://dx.doi.org/10.1016/S0378-7206(01)00091-X`.

BuildAlpha. BuildAlpha Demos (representative screenshot taken from the application), 2022. URL https://www.buildalpha.com/demo/. Retrieved at 25.02.2022.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1. aufl. edition, 8 1996. ISBN 9780471958697.

E.P. Chan. *Machine Trading: Deploying Computer Algorithms to Conquer the Markets*. Wiley Trading. Wiley, 2017. ISBN 9781119219606.

Robert W. Colby. *The Encyclopedia Of Technical Market Indicators, Second Edition*. McGraw-Hill, 2 edition, 10 2002. ISBN 9780070120570.

C. Conlan. *Automated Trading with R: Quantitative Research and Platform Development*. Apress, 2016. ISBN 9781484221785.

J. Danielsson. *Financial Risk Forecasting: The Theory and Practice of Forecasting Market Risk with Implementation in R and Matlab*. The Wiley Finance Series. Wiley, 2011. ISBN 9780470669433. URL https://www.financialriskforecasting.com/book-code/. Retrieved at 23.10.2022.

Marcos Lopez de Prado. *Advances in Financial Machine Learning*. Wiley Publishing, 1st edition, 2018. ISBN 1119482089, 9781119482086.

DukascopyBankSA. Historical Data Export, 2022a. URL https://www.dukascopy.com/trading-tools/widgets/quotes/historical_data_feed. Retrieved at 26.08.2022.

DukascopyBankSA. JForex Platform and API, 2022b. URL https://www.dukascopy.com/swiss/english/forex/jforex/. Retrieved at 20.02.2022.

J.F. Ehlers. *Cycle Analytics for Traders*. Wiley Trading. Wiley, 2013. ISBN 9781118728512.

G. Ford. *Systems Trading for Spread Betting: An End-to-end Guide for Developing Spread Betting Systems*. Harriman House, 2008. ISBN 9781905641734.

H. Georgakopoulos. *Quantitative Trading with R: Understanding Mathematical and Computational Tools from a Quant's Perspective*. Palgrave Macmillan US, 2015. ISBN 9781137354075.

Manolis Georgioudakis and Vagelis Plevris. A comparative study of differential evolution variants in constrained structural optimization. *Frontiers in Built Environment*, 6:102, 2020.

Rozaida Ghazali. Higher order neural networks for financial time series prediction. 2007.

A Hafiiak, E Borodina, and A Diachenko-Bohun. Application of genetic programming tools as a means of solving optimization problems. *Control, navigation and communication systems. Collection of scientific papers*, 6(52):58–60, 2018.

Carol Hargreaves and Chandrika Mani. The selection of winning stocks using principal component analysis. *American Journal of Marketing Research*, 1:183–188, 08 2015.

Roy L Hayes, Peter A Beling, and William T Scherer. Action-based feature representation for reverse engineering trading strategies. *Environment Systems and Decisions*, 33(3):413–426, 2013.

HetznerOnlineGmbH. Dedicated Root Server Hosting, 2022. URL https://www.hetzner.com/dedicated-rootserver/matrix-ex. Retrieved at 09.04.2022.

S. Jansen. *Machine Learning for Algorithmic Trading: Predictive Models to Extract Signals from Market and Alternative Data for Systematic Trading Strategies with Python*. Packt Publishing, 2020. ISBN 9781839217715.

Seth Jennings. Transparent Memory Compression in Linux, 2013. URL https://events.static.linuxfound.org/sites/events/files/slides/tmc_sjennings_linuxcon2013.pdf. Retrieved at 16.09.2022.

JuliaLangContributors. Julia Micro-Benchmarks, 2022. URL https://julialang.org/benchmarks/. Retrieved at 23.10.2022.

Andrew Kumiega and Benjamin Edward Van Vliet. Automated finance: The assumptions and behavioral aspects of algorithmic trading. *Journal of Behavioral Finance*, 13(1):51–55, 2012. ISSN 1542-7560. doi: 10.1080/15427560.2012.654924.

Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. Roaring bitmaps: Implementation of an optimized software library. *CoRR*, abs/1709.07821, 2017. URL http://arxiv.org/abs/1709.07821.

You Liang, Aerambamoorthy Thavaneswaran, Na Yu, Md. Erfanul Hoque, and Ruppa K. Thulasiram. Dynamic data science applications in optimal profit algorithmic trading. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1314–1319, 2020. doi: 10.1109/COMPSAC48688.2020.00-74.

Dome Lohpetch. Evolutionary algorithms for financial trading. 2011. URL https://www.ros.hw.ac.uk/handle/10399/2510.

Alison Lui and George William Lamb. Artificial intelligence and augmented intelligence collaboration: regaining trust and confidence in the financial sector. *Information & Communications Technology Law*, 27(3):267–283, 2018. ISSN 1360-0834. doi: 10.1080/13600834.2018.1488659.

Linkai Luo and Xi Chen. Integrating piecewise linear representation and weighted support vector machine for stock trading signal prediction. *Applied Soft Computing*, 13(2):806–816, 2013.

Pranit Mahajan, Yagnesh Salian, Vinayak Jadhav, and Sujata Kulkarni. A multi-strategic approach to automated trading. In *2021 International Conference on Communication information and Computing Technology (ICCICT)*, pp. 1–7. IEEE, 2021.

James Mayfield, Tim Finin, and Marty Hall. Using automatic memoization as a software engineering tool in real-world ai systems. In *Proceedings the 11th Conference on Artificial Intelligence for Applications*, pp. 87–93. IEEE, 1995.

Nguyet Nguyen. Hidden Markov model for stock trading. *International Journal of Financial Studies*, 6(2): 36, 2018.

oPgroupGermanyGmbH. Zorro Manual: Workshop 5 - Counter Trend, 2022a. URL https://zorro-project.com/manual/en/tutorial_fisher.htm. Retrieved at 20.03.2022.

oPgroupGermanyGmbH. New Functions and Features | Zorro Project, 2022b. URL https://manual.zorro-project.com/new.htm. Retrieved at 16.09.2022.

R. Pardo. *The Evaluation and Optimization of Trading Strategies*. Wiley Trading. Wiley, 2011. ISBN 9781118045053.

K. Price, R.M. Storn, and J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer, 2005. ISBN 9783540209508.

Jonathan Millard Ragan-Kelley. *Decoupling algorithms from the organization of computation for high performance image processing*. PhD thesis, Massachusetts Institute of Technology, 2014.

Schenker Technologies. XMG ULTRA - Gaming Laptops, 2022. URL https://www.xmg.gg/xmg-ultra/. Retrieved at 09.04.2022.

Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

StrategyQuant. StrategyQuant X User's Guide, 2022. URL `https://www.strategyquant.com/licenses/d?code=sqxug`. Retrieved at 25.02.2022.

Pedro Vergel Eleuterio and Lovjit Thukral. Programming language choices for algo traders: The case of pairs trading. *Computational Economics*, 53(4):1443–1449, 2019.

Larry D. Wall. Some financial regulatory implications of artificial intelligence. *Journal of Economics and Business*, 100:55–63, 2018. ISSN 0148-6195. doi: 10.1016/j.jeconbus.2018.05.003.

Muh-Cherng Wu, Sheng-Yu Lin, and Chia-Hsin Lin. An effective application of decision tree to stock trading. *Expert Systems with Applications*, 31(2):270 – 274, 2006. ISSN 0957-4174.

Peter Zwag. GeneticSystemBuilder, 2020. URL `https://trademaid.info`. Retrieved at 17.04.2020.