An AI system to help scientists write expert-level empirical software

Eser Aygün[1,*], Anastasiya Belyaeva[2,*], Gheorghe Comanici[1,*], Marc Coram[2,*], Hao Cui[2,*], Jake Garrison[3,*], Renee Johnston[2,*], Anton Kast[2,*], Cory Y. McLean[2,*], Peter Norgaard[2,*], Zahra Shamsi[2,*], David Smalling[1,*], James Thompson[2,*], Subhashini Venugopalan[2,*], Brian P. Williams[2,*], Chujun He[2,4,**], Sarah Martinson[2,5,**], Martyna Plomecka[2,6,**], Lai Wei[2], Yuchen Zhou[2], Qian-Ze Zhu[2,5,**], Matthew Abraham[2], Erica Brand[2], Anna Bulanova[1], Jeffrey A. Cardille[2,7], Chris Co[2], Scott Ellsworth[2], Grace Joseph[2], Malcolm Kane[2], Ryan Krueger[2,5,**], Johan Kartiwa[2], Dan Liebling[2], Jan-Matthis Lueckmann[2], Paul Raccuglia[2], Xuefei (Julie) Wang[2,8,**], Katherine Chou[2], James Manyika[2], Yossi Matias[2], John C. Platt[2], Lizzie Dorfman[2], Shibl Mourad[1,‡] and Michael P. Brenner[2,5,‡]

[1]Google DeepMind, [2]Google Research, [3]Google Platforms and Devices, [4]Massachusetts Institute of Technology, [5]School of Engineering and Applied Sciences, Harvard University, [6]Google Cloud, [7]Faculty of Agricultural and Environmental Sciences, McGill University, [8]California Institute of Technology

The cycle of scientific discovery is frequently bottlenecked by the slow, manual creation of software to support computational experiments. To address this, we present an AI system that creates expert-level scientific software whose goal is to maximize a quality metric. The system uses a Large Language Model (LLM) and Tree Search (TS) to systematically improve the quality metric and intelligently navigate the large space of possible solutions. The system achieves expert-level results when it explores and integrates complex research ideas from external sources. The effectiveness of tree search is demonstrated across a wide range of benchmarks. In bioinformatics, it discovered 40 novel methods for single-cell data analysis that outperformed the top human-developed methods on a public leaderboard. In epidemiology, it generated 14 models that outperformed the CDC ensemble and all other individual models for forecasting COVID-19 hospitalizations. Our method also produced state-of-the-art software for geospatial analysis, neural activity prediction in zebrafish, time series forecasting and numerical solution of integrals. By devising and implementing novel solutions to diverse tasks, the system represents a significant step towards accelerating scientific progress.

Keywords: Tree Search, Generative AI, Scorable Scientific Tasks, Empirical Software

Introduction

Scientists need diverse information to advance their scientific agendas. Some are simple questions for which perfunctory answers can be fulfilled by a search engine. However, performing computational experiments often demands deeper information. For example, one of the authors' research involves deforestation analyses, assessing land cover change[1] using global spatially-resolved measurements, past and present. This is carried out using a satellite-based deforestation detector, built with code to answer a scientific question. A deforestation detector is one of many thousands of examples of *empirical software* in science. We use the term empirical software to mean software that is designed to maximize a definable or measurable quality score, typically a fit to existing observations. If a task can be solved with empirical software, we call this a *scorable task*.

We have two hypotheses about the scorable tasks and empirical software in science. First, *scorable tasks are ubiquitous in science.* Almost every sub-field of science, applied mathematics, and engineering now relies on software. In the combined experience of the authors, we have found that much of this software is empirical software solving a scorable task. Often such empirical software is at the heart of

[*]Equal contribution in alphabetical order.
[**] Carried out as part of a student researchership at Google Research.
‡ To whom correspondence should be addressed: shibl@google.com, mbrenner@google.com

a scientist's work. Empirical software has recently enabled a number of Nobel Prizes in Chemistry: in 1998 for Density Functional Theory[2,3], in 2013 for molecular dynamics simulation[4] and in 2024 for protein structure prediction[5,6]. Empirical software underlies our ability to create models of complex systems, ranging from parameterizations of a vertical column of the earth's atmosphere for weather modeling[7], to the parameterization of stress response in a turbulent fluid flow[8], to the prediction of social systems[9–11].

Second, *empirical software for science is slow and difficult to create*. Domain-specific empirical software requires tedious work, often over many years. When empirical software is used to test complex hypotheses, it becomes ever more difficult to write purely from first principles. There usually is no systematic search for alternative approaches. Design choices are often governed by intuition or expediency, rather than exhaustive experimentation. Creating the software is so time-consuming that it severely limits the possibilities that can be productively explored.

This paper presents an AI-based system that systematically and automatically creates empirical software to solve scorable tasks. Our method is based on an LLM that rewrites software to attempt to improve its quality score. The system creates a number of software candidate solutions, and uses Tree Search[12,13] to decide which candidates merit further exploration (Fig. 1a). While there are many ways of designing a code mutation system[14–18], we developed and refined the method by designing and competing against a benchmark of basic Kaggle competitions (Fig. 1b), described below. We augment code mutation with research ideas, obtained from a range of sources from highly cited papers, to specialized textbooks, to results of search engines (Fig. 1c). In practice, these ideas can be injected either directly by the user or automatically using a search engine to access research in the literature. The LLM uses this injected guidance in writing code.

We find that our method can be applied to a wide variety of scorable tasks from across science, producing software that outperforms the state-of-the-art produced by scientists. This superhuman performance arises because of the ability to exhaustively and tirelessly carry out solution searches at an unprecedented scale, identifying needle-in-the-haystack high quality solutions.

Results

Overview of Scorable Tasks

We develop our method on a benchmark of Kaggle playground competitions, and test it by selecting scorable tasks based on scientific or engineering problems. We selected these problems using two criteria: first, we chose tasks which have had slow recent progress, but yet are important to a set of scientists; second, we chose tasks which would be useful to the scientific agenda of at least one co-author. These scorable tasks are listed below.

scRNA-seq batch integration:[19] By removing confounding factors, we can enable large-scale multi-lab transcriptomic data integration, such as the Human Cell Atlas[20]. This is a difficult problem because it requires distinguishing subtle biological signals from noise in high-dimensional sparse datasets.

CDC COVID Forecasting:[21] By predicting COVID cases several weeks in advance, we can inform public health policy and resource allocations. The challenge in this task arises from predicting non-linear disease dynamics from lagged and noisy real-time data.

DLRSD segmentation:[22] This is a problem of performing dense pixel-wise multi-label semantic segmentation on complex satellite imagery. Solving this problem can lead to large improvements in environmental monitoring and disaster response.

ZAPBench:[23] This benchmark requires modeling and predicting the activity of >70,000 neurons

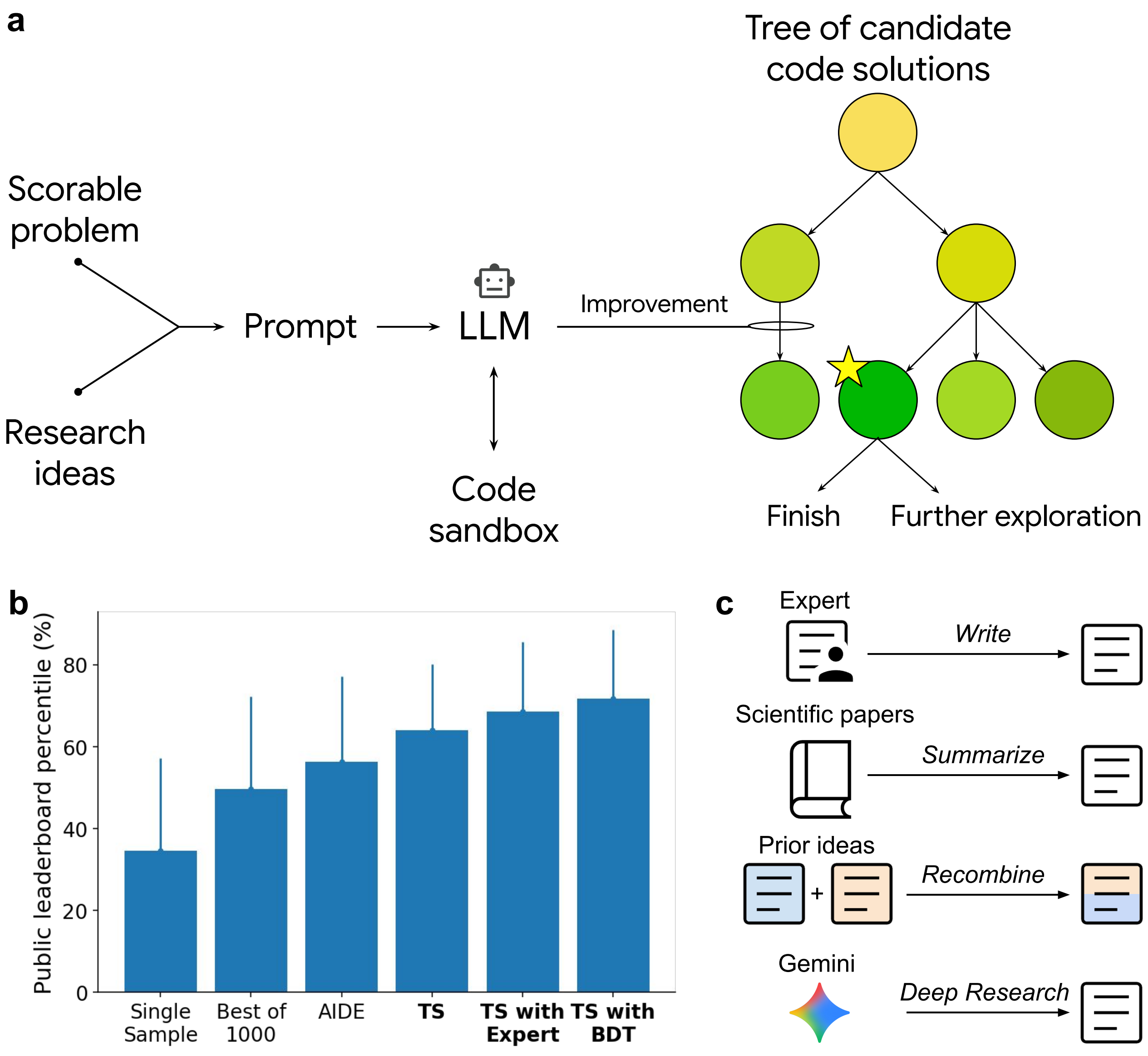


Figure 1 | Schematic and performance of our method. **a**, Schematic of our method algorithm. A scorable task, together with research ideas proposing methods to solve the task, are fed to an LLM, which produces code to evaluate the scorable task in a sandbox. This is then embedded within a tree search algorithm, whereby new nodes are chosen balancing exploitation and exploration, sampling from the LLM (Methods). **b**, Performance of code generation methods on Kaggle Playground benchmark. Results report the average public leaderboard percentile performance over 16 tasks. Methods based on our method are listed in bold. Error bars indicate standard deviation. BDT, boosted decision tree. **c**, Mechanisms used to create initial research ideas to solve scientific problems.

across an entire vertebrate brain. Performing well on this benchmark may lead to a systems-level understanding of brain function and behavior.

GIFT-Eval time series:[24] Accurate time series forecasting is useful for climatology and healthcare. General time series forecasting is very difficult, because of the diverse input feature semantics and prediction time scales. An even more difficult and useful problem is zero-shot prediction, where only an single time series is given and a prediction must be made.

Numerically solving difficult integrals: Solving integrals that defy standard numerical algorithms is useful for modeling physical and engineering systems.

Kaggle Playground Benchmark

We designed our code mutation system to score highly on a curated set of Kaggle competitions. Kaggle calibrates human performance with percentile rank on a leaderboard, and we score code by submitting directly to Kaggle. Our benchmark consists of 16 playground competitions from the 2023 season, encompassing regression and classification tasks. Playground competitions are an ideal benchmark because they offer fast iteration, simplicity, and calibration against thousands of humans. Achieving a high score requires creating complex code without requiring solving a sophisticated scientific task.

Our basic strategy uses a simple prompting template (Supplementary Table 1) that concatenates the competition description with the previous trial. Fig. 1b evaluates the performance of our method with the average public percentile rank across all 16 playground competitions: TS substantially beats a single LLM call and best of 1000 LLM calls. During the search, the agent discovers strategies leading to abrupt jumps in the score, with the accumulation of these jumps leading to the highest quality solutions.

Problem-specific advice added to the prompt substantially improves performance. We illustrate this with two examples. In *TS with expert advice* we give the model standard advice to win Kaggle competitions (Supplementary Table 2). In *TS with Boosted Decision Tree (BDT)* we tell the model to implement a boosted decision tree library from scratch, without using standard packages (Supplementary Table 3). We manually verified in both cases that resulting codes followed the advice.

We now describe evaluating our method on a series of six benchmarks in different scientific fields, exploring distinct ways to incorporate research ideas to improve system performance (Fig. 1c, Methods).

Genomics: Batch Integration of Single Cell RNA Sequencing Data

We first consider data analysis from single cell RNA sequencing (scRNA-seq), which has revolutionized our ability to dissect cellular heterogeneity, discover novel cell types, infer gene regulatory networks and developmental trajectories, and improve therapeutic target prioritization[25], enabling hundreds of millions of cells to be individually sequenced within thousands of datasets[20,26–28]. A major challenge required to jointly analyze many disparate datasets is to computationally remove complex batch effects present across samples while preserving the biological signal[29]. Nearly 300 tools exist to perform batch integration of scRNA-seq data[30], and multiple benchmarks have been developed for assessing metrics of batch effect removal and conservation of biological variability[31–34].

To assess the performance of tree search on this task, we used the OpenProblems v2.0.0 batch integration benchmark[34]. As of July 2025, this active benchmark evaluates 15 state-of-the-art methods and 8 control methods on 13 different metrics that quantify both the ability to remove batch effects in the data and retain variability attributable to true biological differences in six CELLxGENE datasets spanning human and mouse[27] (Fig. 2a). To avoid overfitting to the benchmark, we used a separate dataset from CELLxGENE for hill climbing with our method (Methods, Supplementary Fig. 1). For each tree search run, we selected the best solution based on the performance on this training set, and report the performance on the holdout OpenProblems datasets, which contain in total 1,747,937 cells. We prompt the LLM with a description of the single cell batch integration problem, code for reading in the dataset, code for evaluation metrics, and optional text with a particular research idea.

First, we ran tree search without guidance, and observed that its solution is conceptually similar to ComBat[37], yet improved over the current OpenProblems leaderboard (`No advice (TS)` in Fig. 2b). We then evaluated whether our method could improve upon existing algorithms. We selected nine methods from the OpenProblems benchmark, including the six highest-performing methods (Methods). For each method, we obtained the paper PDF and used Gemini 2.5 Pro to add a brief summary to the prompt (Methods). In pairwise comparisons, our method outperformed the corresponding published result for eight of the nine methods in overall score (Fig. 2b, Supplementary Table 4). The top-performing method was our tree search based implementation of Batch Balanced K-Nearest Neighbors (`BBKNN (TS)`)[38], yielding a 14% overall improvement over the best published method (`ComBat`[37]) and equaled or outperformed the corresponding published `BBKNN` in every dataset and across 11/13 metrics (Fig. 2b). This performance highlights its capacity to effectively remove batch effects without compromising biological signals (Supplementary Fig. 2). We note that tree search is also able to produce performant implementations for an algorithm without publicly-available code (TabVI[39], Supplementary Fig. 3). Importantly, expert manual inspection of the code solutions proposed by our method confirmed that nearly all implementations adhered to the requested algorithms (Supplementary Table 5), with performance largely consistent across replicate runs of methods (Supplementary Fig. 3). Additionally, tree search demonstrated improvements even when compared to base methods with optimized hyperparameters, indicating that its contribution extends beyond hyperparameter tuning (Methods, Supplementary Fig. 4). Supplementary Fig. 5 shows representative examples of the tree structure and breakthrough plots (showing the evolution of the maximum score as a function of the number of nodes in the tree) for a representative example.

For the best performing model `BBKNN (TS)`, part of the performance boost came from combining two existing methods, ComBat[37] and BBKNN, rather than simply implementing BBKNN (Fig. 2c). In particular, while the original BBKNN method computes neighbors on the PCA embedding, `BBKNN (TS)` computes neighbors on ComBat-corrected PCA embedding, removing global linear batch-associated variance. Both implementations then compute k-nearest neighbors across batches and construct a graph (with differences in exact implementation), thus removing local batch effects. Manual modification of `BBKNN (TS)` and the published BBKNN implementation confirmed that the addition of Combat-corrected PCA embedding is critical for improving both implementations (Supplementary Fig. 6), confirming the value in idea recombination.

This motivated an exploration of systematic ways to generate more complex research ideas. First, similar to how scientists often combine ideas to create a novel approach, we programmatically generated 55 “recombinations” of all pairs of the 11 methods described above (No advice, nine replications, and TabVI; hereafter: “base methods”) based on summaries of the code for each method (Methods, Supplementary Table 6). We ran tree search, prompted with each of these “recombinations” to assess whether it can develop new methods by combining the strengths of the existing methods. For each base method and “recombination” group, we compared the average scores for the top nodes over the intersection of metrics that were successfully computed for all three methods. Strikingly, recombination implementations of tree search frequently outperformed their base counterparts, with 24 of the 55 “recombination” solutions (44%) outperforming both of their base methods and 22 of the remaining 31 “recombination” solutions outperforming one of the two base methods (Supplementary

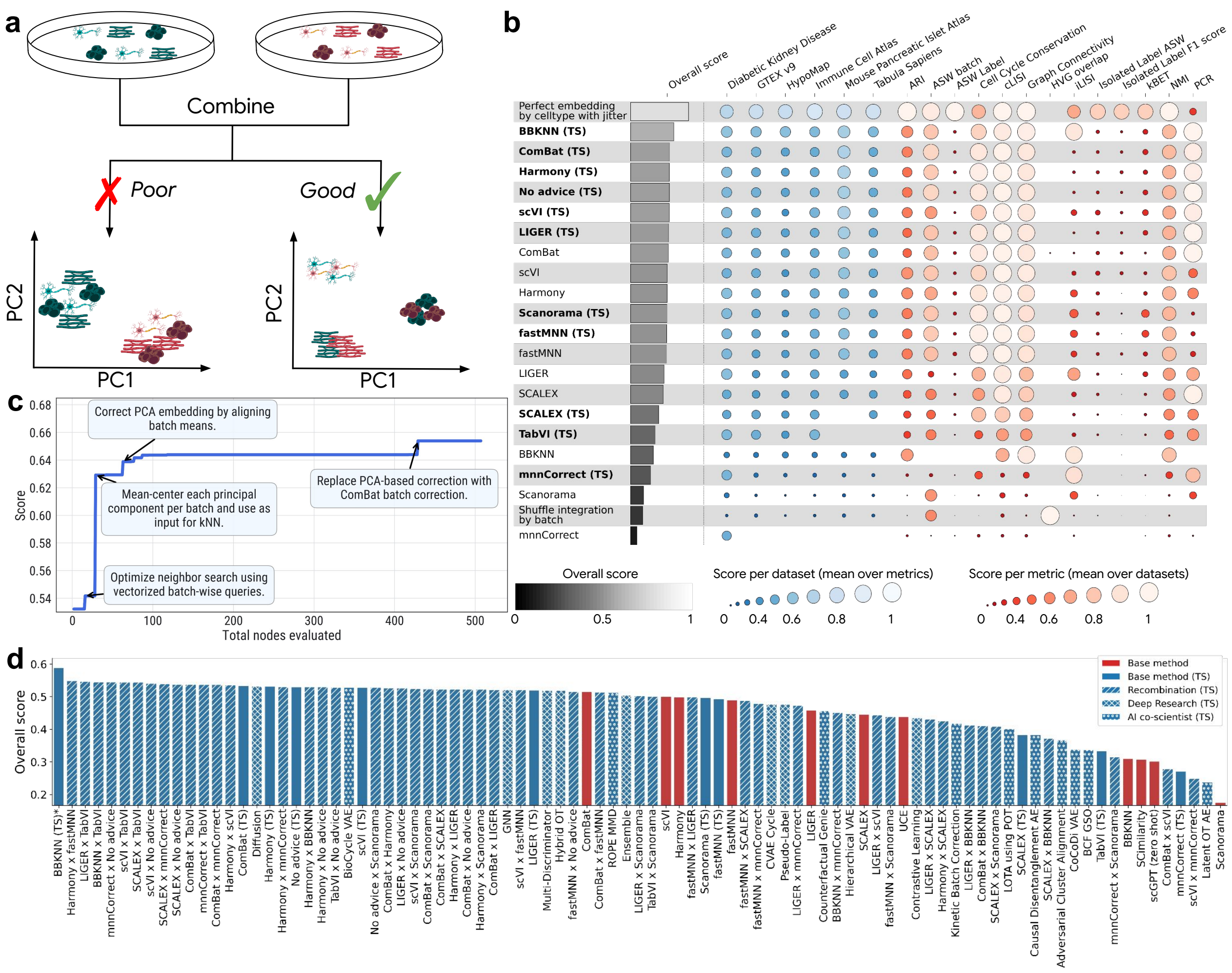


Figure 2 | Performance of tree search on scRNA-seq batch integration. **a**, Schematic of the batch integration task, in which disparate datasets (teal and red) are processed to remove batch effects in the data while retaining biological variability. **b**, Performance of tree search (method names bolded and suffixed by "(TS)") compared to the analogous published method on the OpenProblems benchmark v2.0.0[34]. "Perfect embedding by celltype with jitter" is a positive control method that represents the best possible performance and "Shuffle integration by batch" is a negative control that does not perform any batch integration. Overall score is the mean over all datasets and metrics. Each Datasets column shows the mean of all metrics computed over that dataset. Each Metrics column shows the mean of that metric computed over all datasets. Metrics were assigned a value of 0 if they could not be computed or if their performance was worse than the lowest negative control; these are displayed as empty. **c**, Performance improvements annotated with code innovation for the top-performing batch balanced *k*-nearest neighbors (BBKNN) implementation. ComBat-based embedding generation was introduced in implementation attempt 429. **d**, Overall score for OpenProblems benchmark v2.0.0[34] non-control methods, our method with and without recombination of ideas, Gemini Deep Research[35], and our method with AI co-scientist[36]. Y-axis lower bound is the overall score of the "Shuffle integration by batch" negative control method. Seven recombination, five base methods, and two AI co-scientist methods that do not match its performance are omitted. * indicates the method is a recombination, even if not explicitly prompted for recombination. TS, tree search; fastMNN, batchelor fastMNN; mnnCorrect, batchelor mnnCorrect.

Fig. 7). Second, we also used Gemini Deep Research[35] and AI co-scientist[36] to generate and implement 21 additional ideas (Methods). In total, 6/11 base methods, 29/55 recombination, 4/9 Deep Research, and 1/12 AI co-scientist methods (40 of 87) outperform all methods currently published on the OpenProblems leaderboard (Fig. 2d). This demonstrates the ability of our method to understand the best features of existing approaches and effectively integrate them for superior performance.

To further understand the conceptual space explored by our method, we obtained embeddings for each generated code using Gemini text embedding model and computed cosine similarities (Supplementary Fig. 8). As expected, replicates exhibited significantly higher similarity to each other compared to all other method pairs (one-sided t-test: $t = 12.95$, $p = 1.06 \times 10^{-14}$; $\mu_{\text{duplicate pairs}} = 0.95$, $\mu_{\text{other pairs}} = 0.91$; $n_{\text{duplicate pairs}} = 33$, $n_{\text{other pairs}} = 5853$). Hierarchical clustering on the embeddings revealed distinct clusters, generally representing linear methods, deep learning based methods, and nonlinear non-deep learning methods, suggesting that our method is able to generate diverse solutions.

Public Health: Prediction of U.S. COVID-19 Hospitalizations

The primary U.S. benchmark for COVID-19 forecasting is the COVID-19 Forecast Hub (CovidHub)[21], a large, collaborative effort coordinated by the Centers for Disease Control and Prevention (CDC). The hub attracts dozens of expert-led teams from leading academic institutions, industry, and government agencies, who submit weekly forecasts generated from a wide array of methodologies. These weekly forecasts must cover new COVID-19 related hospitalizations across 52 U.S. states and territories for the current week and three subsequent weeks over 23 specified quantiles. Submissions are evaluated using the Weighted Interval Score (WIS), which rewards both accuracy and well-calibrated uncertainty, with lower scores indicating better performance.

Top-performing individual models include classic autoregressive time-series approaches (e.g., UMASS-ar6_pooled), gradient boosting machine learning models (e.g., UMASS-gbqr), and epidemiological models based on renewal equations and Bayesian estimation of the reproductive number (e.g., CEPH-Rtrend_covid). The hub leverages this methodological diversity by integrating submissions into the CovidHub Ensemble, a robust aggregate forecast that has historically provided the gold standard for epidemiological prediction in the U.S., making it a formidable benchmark to outperform.

We designed a rigorous retrospective study to assess tree search's performance in this competitive environment. For every forecasting period, we ran tree search to optimize and select a model using data from the preceding six weeks, creating a rolling validation window throughout the 2024-2025 season (Fig. 3a). The weekly performance of our resulting 'Google Retrospective' model is detailed in the time-series leaderboard (Fig. 3b), which visualizes our model's performance advantage relative to the CovidHub-ensemble and other top-performing teams. Supplementary Fig. 9 shows the temporal variation of WIS for each of the separate validation splits, across replicates Supplementary Fig. 10. A direct jurisdiction-level comparison confirms our model achieved a lower (better) WIS in a majority of states (Fig. 3c), with the geographic distribution of performance shown in Fig. 3d. Overall, our model achieved the highest performance with an average WIS of 26, outperforming the official CovidHub Ensemble's average WIS of 29. A representative tree and breakthrough plot is shown in Supplementary Fig. 11.

Beyond this retrospective performance, we investigated our method's ability to explore the solution space more broadly by replicating, recombining, and generating entirely new forecasting strategies (Fig. 3e). First, we tested its ability to replicate existing methods from other teams using only their brief public descriptions from the CovidHub (Supplementary Table 7,Supplementary Table 8). Our tree-search-based implementations ('Base Method (TS)') not only adhered to the provided instructions

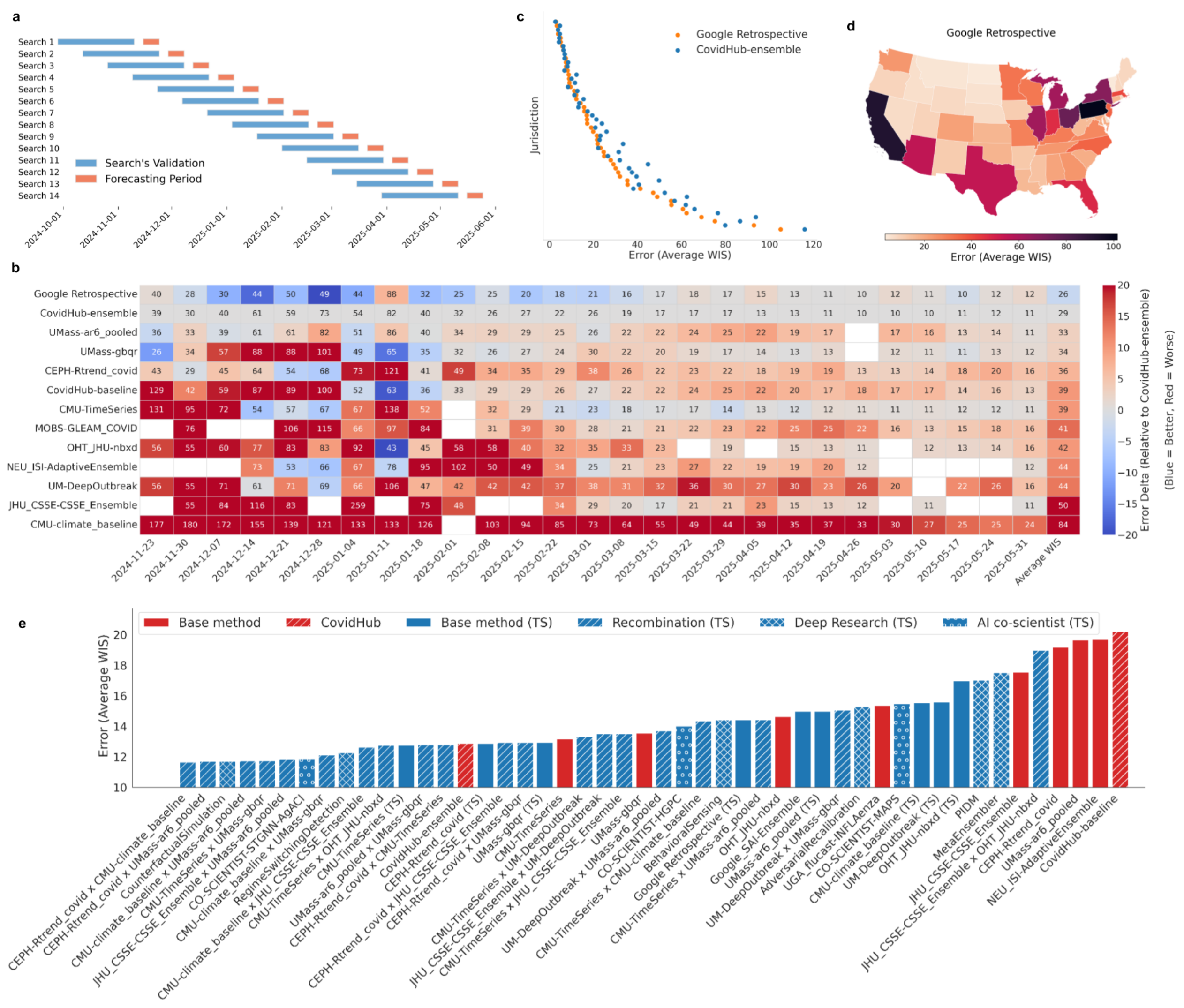


Figure 3 | Performance of tree search on COVID-19 forecasting. a, Rolling validation window used for the forecasting experiments. Each search's output is validated internally on a preceding block of time (blue), and the resulting model is then used to make predictions for its corresponding forecasting period (orange). Training data includes all dates on or after 2020-08-08 and prior to the validation set. **b,** Time-series leaderboard showing weekly forecasting performance (Average WIS) for participating teams and our 'Google Retrospective' model, ordered by average WIS. Scores are aggregated across all 52 jurisdictions and four forecast horizons. The number within each cell is the model's absolute Average WIS for that week. The cell's background color visualizes the performance relative to the CovidHub-ensemble, with blue indicating a lower (better) WIS and red indicating a higher (worse) WIS. **c,** Direct jurisdiction-level comparison of forecasting error (Average WIS) between our model and the 'CovidHub-ensemble', demonstrating our model's superior performance in a majority of locations. **d,** Geographic distribution of our model's forecasting error (Average WIS), aggregated over the entire 2024/25 COVID-19 season. Lower error values (lighter colors) indicate better performance. **e,** Comparison of aggregate forecasting performance for various modeling strategies. This includes baseline models from the CovidHub competition, our retrospective model, our replications of submitted models, novel hybrid models generated through recombination, deep research[35] and AI co-scientist[36]. 14 strategies (10 recombination; two Deep Research; one AI co-scientist and one replicated baseline) outperform the official CovidHub-ensemble for the 3-week (3 reference dates × 4 time horizons × 52 jurisdictions) evaluation period. Models that perform worse than CovidHub-baseline are not shown.

(Supplementary Table 9) but also exceeded the performance of the original submissions in six of the eight cases tested; the two models that performed worse (replicating `JHU_CSSE-CSSE_Ensemble` and `OHT_JHU-nbxd`) did not use external data present for the original method implementations. Next, we explored whether solutions could be improved through recombination. For this experiment, we prompted an LLM to analyze the core principles of two different parent models and then used its synthesis to instruct tree search to generate a novel hybrid strategy combining their respective strengths. As shown in Fig. 3e, 11 out of 28 generated hybrid models ('Recombination (TS)') achieved a WIS score superior to both of their parent models (Supplementary Fig. 12). We manually verified methodology of the output code for the recombined experiments–in all cases, the final methods contained relevant aspects from both parent codes (Supplementary Table 9). Finally, we used Gemini Deep Research[35] and AI co-scientist[36] to generate novel forecasting ideas which were then implemented via tree search. In total, this systematic exploration yielded 14 distinct strategies that outperformed the official CovidHub-ensemble: 10 from recombination, two from Deep Research, one from AI co-scientist, and one of our replicated baselines. Cosine similarities between embeddings for each generated code show clustering between different methods (Supplementary Fig. 13).

A deeper analysis of these 14 top-performing strategies reveals key patterns in how our method achieves superior performance. The recombination models, which constitute the majority of the winners, highlight a clear pattern of synergistic hybridization. Two base models appear most frequently in these successful hybrids: the simple, climatology-based `CMU-climate_baseline` and the statistical autoregressive model `UMass-ar6_pooled`. This suggests tree search consistently discovers that the most effective strategies are built upon a robust foundation of historical averages and recent trends, which are then enhanced by more complex methods. Indeed, the most successful recombinations consistently fused different modeling paradigms—for instance, pairing the epidemiological `CEPH-Rtrend_covid` model with the statistical `UMass-ar6_pooled` model created a hybrid anchored in the theory of disease spread yet highly responsive to recent data trends, while pairing the powerful machine learning `UMass-gbqr` model with the stable `CMU-climate_baseline` provided a robust seasonal foundation that allowed the ML model to safely focus on learning short-term deviations—demonstrating an ability to synthesize complementary strengths.

In contrast, the novel strategies generated via Deep Research and AI co-scientist represent significant conceptual leaps beyond the existing Hub models. Rather than relying on conditional uncertainty from past data, the `DEEP-RESEARCH-CounterfactualSimulation` model introduces unconditional uncertainty quantification by running thousands of Monte Carlo simulations over plausible future scenarios (e.g., new variant emergence). Similarly, while some base models use deep learning, the `CO-SCIENTIST-STGNN-AgACI` model implements a far more complex Spatio-Temporal Graph Neural Network with a learnable graph structure to explicitly model inter-state dynamics. The `DEEP-RESEARCH-RegimeSwitchingDetection` model introduces another novel concept: dynamic, event-triggered adaptation, using Bayesian change-point detection to automatically initiate model retraining in response to shifts in the underlying data generating process. Finally, the outperformance of our replicated `CMU-TimeSeries (TS)` model underscores that even when not inventing or hybridizing, tree search excels at the fine-grained optimization of already-strong, expert-designed strategies. Ultimately, this demonstrates the power of tree search as a scientific discovery engine, capable of systematically exploring a vast solution space to innovate, hybridize, and optimize expert-level strategies.

Geospatial Analysis: Segmentation of Remote Sensing Images

We now turn to a problem in geospatial analysis: semantic segmentation of high-resolution remote sensing images. Semantic segmentation is a computer vision task that involves assigning a specific class label to every single pixel in an image. It is essential for diverse applications, ranging from monitoring land use, assessing the environmental impacts of human activity and managing natural disasters. The primary difficulty is significant visual heterogeneity. Satellite images of the same location can differ dramatically due to variations in time of day, season, and weather conditions, while even objects within a single class (e.g. buildings) exhibit substantial diversity in size, shape, height, function and lighting conditions.

A recent paper[22] introduces the "dense labeling remote sensing dataset" (DLRSD) for advanced remote sensing tasks, including multi-label classification, image retrieval, and pixel-based applications like semantic segmentation. This dataset is a densely labeled version of the UC Merced Land Use Dataset[40], a widely-used benchmark for image-level land use classification, whereby individual pixels of each image are labeled with 17 class labels.

We prompted our method to train a model to classify pixels into the land cover classes and provided a pre-specified, reproducible 80/20 train/test split of imagery in the DLRSD dataset. For each experiment, we validated model performance on the held out test set of 420 randomly selected images using a standard "mean intersection over union" (mIoU) metric.

The three top performing solutions generated by tree search significantly outperformed reported results in recent academic papers on the DLRSD benchmark, achieving mIoU greater than 0.80 (Table 1, Supplementary Fig. 14). All three solutions build upon existing models, libraries and strategies. Solutions 1 and 3 leverage standard UNet++ and U-Net models but paired with powerful encoders (efficientnet-b7 and se-resnext101-32x4d) pre-trained on ImageNet[41]. Solution 2 uses SegFormer, a state of the art Transformer-based architecture. Key differentiators among the models included their data augmentation and prediction strategies. The U-Net++ and U-Net models leveraged extensive augmentation from the Albumentations library, whereas the Segformer model used a more basic set of transforms. All three solutions employ extensive Test-Time Augmentation (TTA)[42] by predicting masks for multiple augmented versions of a single test image (e.g., horizontal flips, vertical flips, rotations) which are then reverse-transformed and averaged to produce a final, more robust mask which smooths out prediction errors and boosts performance. A representative tree and breakthrough plot for Solution 3 is shown in Supplementary Fig. 15.

Neuroscience: Whole-Brain Neural Activity Prediction

We now consider the Zebrafish Activity Prediction Benchmark (ZAPBench), a recent dataset designed to test predictions of cellular-resolution neural activity in an entire vertebrate brain[23]. The benchmark uses a novel dataset capturing brain activity of a larval zebrafish over a two-hour session using light-sheet fluorescent microscopy, resulting in 3D brain volumes recorded over time. Throughout the recording, the animal was exposed to distinct visual stimulus conditions designed to elicit a range of different behaviors. The raw volumetric video data was extensively processed to align, motion-stabilize, and segment into activity traces, resulting in a final data matrix of activity traces for 71,721 neurons across 7,879 time steps.

Several state-of-the-art forecasting methods were evaluated on the benchmark[23], including time-series forecasting methods that operate on the extracted activity traces per neuron, as well as a volumetric video prediction model (a Unet variant) that directly processes the 3D brain volumes over time[48]. The video-based approach exploits spatial information that is lost when converting the data to time series, but is computationally expensive. Among the different methods evaluated on the benchmark, the video-based Unet model achieved the best overall performance, especially in the setting where only a short window of past context is available.

We prompted our method to solve the multivariate time-series forecasting problem, predicting

Table 1 | Comparison of model performance on the DLRSD benchmark. The table shows the publication year, architecture, key features, and reported mean Intersection over Union (mIoU) for tree search solutions and the methods from the referenced papers.

Method	Year	Architecture Type	Key Features / Techniques	mIoU
Solution 1 (TS)	2025	CNN (UNet++)	'efficientnet-b7' encoder, 8-fold TTA	0.81
Solution 2 (TS)	2025	Transformer(SegFormer)	'mit-b1' encoder, 4-fold TTA	0.82
Solution 3 (TS)	2025	CNN (U-Net)	'se_resnext101_32x4d' encoder, 7-fold TTA	0.80
RE-Net[43]	2021	CNN (Region-based)	Region Context Learning	0.762
FURSformer[44]	2023	CNN+Transformer	Custom fusion module	0.753
SCGLU-Net[45]	2024	CNN+Attention	Spatial-Channel-Global-Local block	0.666
MA-UNet[46]	2022	Attention+U-Net	Residual encoder with simAM	0.619
W13 Net[47]	2025	CNN (Lightweight)	Multi-stage encoding-decoding	0.580

the output activity of all neurons for up to 32 time steps ahead in the time-series domain, given their past 4 time steps of activity as context, using the dataset splits provided by ZAPBench[23] which split each stimulus condition into 70% for training, 10% for validation, and 20% for testing per stimulus condition. We used the validation set for model selection, including hyperparameter tuning and early stopping, and to obtain a score to guide the tree search. We score solutions using mean absolute error (MAE) averaged across the prediction horizon, and compare solutions found by tree search against the methods included in ZAPBench: These include a linear model[49], TiDE[50], TSMixer[51], Time-Mix (a variant of TSMixer where feature mixing is ablated), and a custom Unet architecture[48].

Our initial experiment using tree search led to a best-performing model that uses a rich feature set from the input window, combining temporal convolutions, a learned "global brain state", and neuron-specific embeddings. The model then processes these features through a series of weight-shared residual blocks and a final dense layer to generate the multi-step prediction in one shot. Figure 4 shows the result of this model, compared to other baselines. In that figure, the mean baseline predicts the average over the context window, while the stimulus baseline predicts the average for each stimulus phase. Remarkably, the model produced by tree search outperformed all other baselines, including the best-performing video model, except for 1-step-ahead predictions. A representative example of the breakthrough plot and tree is shown in Supplementary Fig. 16.

We then developed a separate model tuned specifically for 1-step-ahead predictions with another tree search. The resulting solution is conceptually similar to the first in that both architectures generate a learned global context vector to inform their per-feature predictions. However, this model computes its global context using a dynamic attention mechanism for weighted aggregation and modulates feature representations through a FiLM-like layer[52] for interactive conditioning. This model achieved leading performance on 1-step-ahead predictions (Fig. 4).

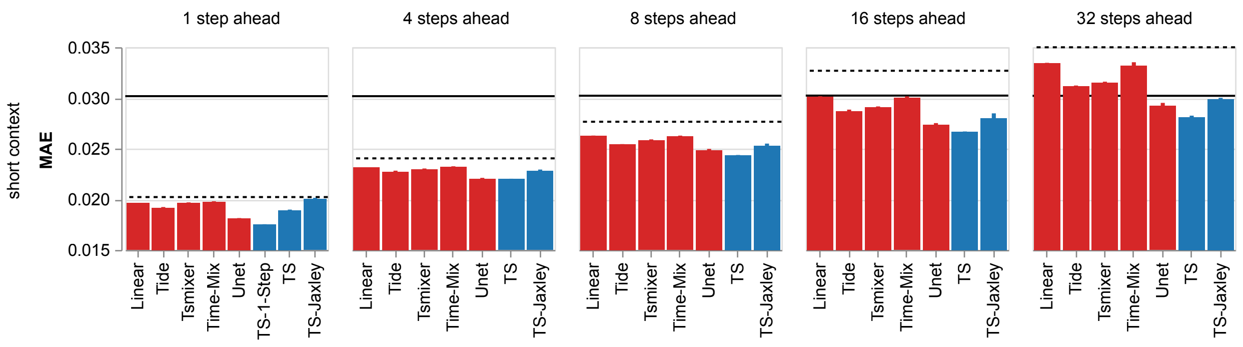


Figure 4 | Comparison of the best tree search solutions to time-series and video forecasting methods in terms of grand average mean absolute error (MAE) across conditions on ZAPBench (lower is better). For our method, we report the performance of three different solutions (blue), and compare them against baselines (red). Alongside our best general solution (TS), we include results from two specialized runs: a tree search that was optimized for 1-step ahead forecasting as well as a solution prompted to use Jaxley, a differentiable biophysical neuron simulator. The dotted and solid lines represent the mean and stimulus baselines, respectively. To account for variability due to random number generator seeding, each method was run three times. We report the mean, with error bars indicating 95% confidence intervals.

Both of these solutions are orders of magnitude faster to train than the best-performing video model–less than two hours on a single T4 GPU, as compared to 36 hours on 16 A100 GPUs for the Unet model. In addition, our solutions effectively use cross-neuron information to generate predictions, a major challenge highlighted in previous work[23].

A key future direction is the development of models that incorporate biophysical information and are more interpretable. The forthcoming synaptic-level structural reconstruction of the larval zebrafish brain used for ZAPBench provides a unique opportunity to develop such models by integrating anatomical wiring diagrams. As an initial exploratory step, we prompted our method to use Jaxley[53], a JAX-based library for differentiable simulation of biophysically detailed neuron models, for the tree search. The resulting best-performing solution simulates each neuron independently using single-compartment Hodgkin-Huxley models. Crucially, it dynamically modulates each neuron's biophysical parameters based on its recent activity history. To account for inter-neuronal interactions without the computational cost of direct synaptic simulation, the model then processes the outputs of these independent simulations through a latent autoencoder. This learns a system-wide corrective signal, effectively modeling a *functional* connectome–a reasonable hybrid approach in the absence of the structural connectome. While this model did not outperform the top-performing video model, it was competitive with time-series baselines (Fig. 4).

Time Series Forecasting: GIFT-Eval

General Time Series Forecasting Model Evaluation (GIFT-Eval)[24] is a benchmark for time series forecasting, derived from 28 datasets from seven diverse domains, with 10 different frequencies, from seconds to years, receiving ~4 new submissions per month, from black box deep learning to foundation models. Submissions are scored on official train/validation/test splits using a normalized Mean Absolute Scaled Error (MASE) metric, calculated relative to a seasonal naive baseline.

We applied our method in two phases. We began with a `per-dataset solution` whereby the search discovers an independent solution for each. The second `unified solution` created a single a general-purpose forecasting model using only basic libraries by hill climbing against the average score for the entire GIFT-Eval.

Per-dataset solution Here we allowed our method to use a full suite of Python libraries, including `scikit-learn`, `statsmodels`, and `xgboost`. The results in Supplementary Table 10 are better than the results in the May 18, 2025 leaderboard, outperforming foundation models[54–56], deep learning models[49,50,57] and standard time series methods like ARIMA[58]. The discovered solutions showed strong convergence towards `gradient boosting` and `ensemble/decomposition` models (Supplementary Fig. 17).

Unified solution We wondered whether the code mutation system could create a unified, general-purpose forecasting library from scratch, by hill climbing with a *single* code on the average MASE on the entire GIFT-Eval dataset. To manage the benchmark's diversity, we allowed library to have an adaptive configuration system, whereby it could generate up to 8 preset hyperparameter configurations to adapt to the diversity of datasets, with a validation step selecting the best performing configuration for each dataset. As the search progressed, date and trend-related features often led to performance breakthroughs leading to a model that sequentially forecasts and subtracts individual time series components, including a base level, trend, seasonality, datetime-based features, and a final residual correction. Supplementary Fig. 18 shows the breakthrough plot and tree structure for the search. The configurations (Supplementary Table 11) include date-specific features, including one that featurizes holidays in a specific set of countries ([‘US’, ‘DE’, ‘CN’, ‘GB’, ‘CA’, ‘AU’]) . The resulting `unified solution` is highly competitive on the leaderboard (Supplementary Table 10).

Numerical Analysis: Library for numerical evaluation of difficult integrals

Finally, we turn to a problem in numerical analysis, the numerical evaluation of difficult integrals using Gaussian quadratures. The gold standard was developed[59] by U.S. government research laboratories in the 1980s, widely used as the core library underlying the popular Python function `scipy.integrate.quad()`. Nonetheless, this function can fail in multiple ways, among them: the underlying algorithm can fail to converge; the algorithm samples its integrand, and the sampling may miss important features; the algorithm loses precision when the problem exhibits precise cancellations.

While standard techniques exist to address these problems, we asked whether our method could build a general-purpose method superior to `quad()`, by hill climbing on a benchmark set of integrals where the standard algorithm fails but where the analytic answer to the integral is known. We constructed this set of integrals from a standard applied mathematical reference book[60], focusing on oscillatory integrals with infinite upper limits and without other pathologies, but where the standard `quad()` library returned an incorrect answer. This led to a set of 38 integrals (Supplementary Fig. 19). We split these in half at random, using 19 for scoring the search and holding out the other 19 for evaluation. We then initialized our method with a simple invocation of `quad()` and prompted the system to improve it, scoring solutions with the logarithm of the absolute fractional error, where the logarithm prevented the search from over-weighting outliers. Supplementary Fig. 20 shows the resulting breakthrough plot and tree structure for the search.

The best solution builds on `quad()` by partitioning the infinite domain into a sequence of contiguous, finite subintervals whose lengths may increase geometrically to cover the domain's tail more efficiently. The definite integral is thus transformed into an infinite series, where each term is the numerical integral of the integrand over one of these finite segments, calculated using `quad()`. For integrals that converge slowly, such as those with oscillatory integrands, direct summation of this series is impractical.

The algorithm therefore applies Euler's transformation, a powerful series acceleration technique, to this sequence of segment integrals. By repeatedly averaging adjacent terms, the transformation extrapolates the limit of the slowly converging series from a finite number of its initial terms, providing an accurate estimate of the integral's true value.

Whereas `scipy.integrate.quad()` fails on every problem in the held-out set, the evolved code correctly evaluated 17 out of 19 of held-out integrals to within a fractional error of less than 3 percent (Supplementary Fig. 21).

The evolved code always applies `scipy.integrate.quad()` first. It only falls back to its more specialized methods if `quad()` returns a large error estimate, returns `NaN` or `Inf`, or raises an exception. This means the evolved code is as accurate as `quad()` in less pathological cases and so could reasonably be used as a drop-in replacement.

Discussion

Our work introduces an AI-based system that drives a Tree Search (TS) with a Large Language Model (LLM) to systematically create and improve software for scientific tasks. By defining the problem of creating scientific software as a search for a program whose output maximizes a quality score, we convert software creation into a "scorable task", producing empirical software. Our method is novel in its LLM-driven rewriting approach, which allows for the flexible integration of domain knowledge and external research ideas. The ability of frontier LLMs to closely follow instructions enables efficient exploration of research ideas. Our method builds upon ideas from several distinct but related areas of research: Genetic Programming, Generative Programming, the application of LLMs to code, Automated Machine Learning (AutoML), and agents for scientific discovery.

Genetic Programming — The idea of automatically evolving computer programs to solve a problem is not new. Genetic Programming (GP) provides a foundation to our work. In GP, a population of programs is iteratively improved using evolutionary principles like selection, crossover, and mutation. The fitness of each program is determined by a "fitness function," which is directly analogous to our "quality score"[61]. While GP has been successful, it traditionally relies on random mutations and structured recombination of code fragments (e.g., swapping sub-trees in an abstract syntax tree). A key difference in our system is the use of an LLM to perform intelligent, semantic-aware "mutations" by rewriting the code, which can produce more complex and meaningful variations than the random changes typical in GP.

Generative Programming — Our system can be viewed as a modern, AI-driven realization of this concept. In traditional generative programming, a developer creates a program generator (using techniques like templates, domain-specific languages[62], or metaprogramming) that produces tailored source code for a family of related problems[63]. In contrast, we employ an LLM guided by a tree search as the generative engine. This approach offers greater flexibility, allowing the system to synthesize novel programs by exploring a vast solution space and integrating diverse domain knowledge in ways not easily achievable with more template-based methods.

LLMs for Code Generation — The advent of large language models pre-trained on vast code corpora has revolutionized code generation. Systems like OpenAI's Codex[64] and Google DeepMind's AlphaCode[65] have demonstrated the ability to generate correct and complex code from natural language descriptions. These systems are typically used for "one-shot" generation from a prompt. Our approach differs by using the LLM in an iterative refinement loop. Instead of generating code from scratch, our LLM rewrites existing software candidates, guided by a search algorithm (TS) that uses the quality score as a signal.

Combining LLMs and Search — The most closely related work involves combining LLMs with search algorithms to overcome the limitations of one-shot generation. A recent example is Google DeepMind's FunSearch, which uses an LLM to search for new mathematical discoveries[16]. FunSearch works by pairing a creative LLM with an automated evaluator. The LLM suggests improvements (new code) to an existing program, and these improvements are only kept if they pass evaluation. This creates an evolutionary feedback loop. This is conceptually very similar to our system's use of an LLM rewriter and a quality score. However, our system generalizes the search process using TS, a robust algorithm for exploring large search trees. Our system also incorporates knowledge from the literature.

AutoML — our work is conceptually related to Automated Machine Learning (AutoML). AutoML systems aim to automate the process of building machine learning pipelines by searching for optimal model architectures and hyperparameters. The goal is to maximize a performance metric (e.g., accuracy, F1-score) on a validation dataset[66], which fits our definition of a scorable task. While AutoML focuses specifically on finding the best model within a fixed set of ML frameworks, our system is more general. It can rewrite any software, including pre-processing steps, complex simulations, or mathematical heuristics—areas that fall outside the typical scope of AutoML.

Agents for science problems — This sub-field has seen remarkable, expert-exceeding performance from highly specialized systems[67]. Much of the existing literature focuses on agents that either automate standard workflows within a single domain, such as computational biology[68–72], or act as ideation assistants whose proposals require significant human validation[73,74]. Instead of specializing in one domain, our system demonstrates a general problem-solving capability, achieving expert-exceeding performance on public leaderboards and in academic literature across multiple fields.

To summarize, we have developed a method that combines a code mutation system based on Tree Search[12,13] with the ability to integrate complex research ideas. Such research ideas could come from the published literature, from research agents (e.g.[35,36,75]) or from combining previous ideas and solutions that the LLM has found itself. Because the system creates code that can follow a specific idea, it can search over externally supplied research ideas. We demonstrate over a wide range of scientific scorable tasks that reaches an expert-level when integrating and exploring complex research ideas.

Our method created 40 methods that beat the best known method for scRNA-seq batch integration and 14 methods that outperformed the CDC ensemble for epidemiological prediction. Additionally, our method achieved state of the art performance on geospatial reasoning, neural activity prediction, time series prediction and algorithms for computational mathematics. With minimal prompting, the system invents and implement ideas for combining complex architectures (U-Nets, transformers) for a fundamental task in geospatial reasoning, and in neural activity prediction it was not only able to outperform all methods on the current benchmark, but easily incorporates a biophysical simulator into a performant solution.

Trial and error is essential to scientific progress, both for humans and for the automated approaches we outline here. The system generates expert-level solutions extraordinarily quickly, reducing exploration of a set of ideas from weeks or months, to hours or days. Accelerating research in this way has profound consequences for scientific advancement. Based on this work, we believe that progress in scientific fields where solutions can be scored by machines is on the precipice of a revolutionary acceleration.

Acknowledgements

We are grateful to our colleagues in Google Research and Google DeepMind for the incredible environment within which to do this work. We would like to specifically thank Niv Efron, Viren Jain, Anupam Pathak and Jamie Smith for many incisive discussions, and Nicholas Reich for comments on the manuscript.

Author Contributions Statement

Code Mutation System (E.A., A.B., G.C., M.C., H.C., P.N., D.S., J.T., S.V., M.P., J.K., P.R., J.W., L. W., S.M. and M.P.B.) Single Cell RNA-seq Batch Integration (A.Be., C.Y.M., C.H., Y.Z., M.P.B.) COVID Forecasting (Z.S., S.M., M.P., M.C., M.P.B.) Geospatial Analysis (R.J., Je.C., Q.Z., M.P.B.) ZAPBench (B.P.W., J-M.L, Q.Z.) GIFT-Eval (J.G., M.P.B.) Integrals (A.K., R.K., M.P.B.) User Interfaces (E.A., G.C., P.N., A.K., M.K., M.P.B., J-M.L., D.L., J.K., C.C., S.E.) Graphical Design (G.J.) Program Management (M.A., E.B.) Leadership (K.C., J.M., Y.M., J.C.P., L.D., S.M., M.P.B.)

Code Availability

We are open sourcing the best candidate solutions generated from each of the examples outlined in this paper (github.com/google-research/score). Additionally, we are providing a user interface to examine the full tree search data for a representative example of each of the six scientific problems discussed in the paper. The interface allows inspecting the solution progression and breakthrough plot as the tree search proceeds, as well as highlighting the code diffs.

Methods

Code Mutation System

We prompt an LLM (Supplementary Fig. 22) providing a description, the evaluation metric and the relevant data. The LLM produces Python code, which is then executed and scored on a sandbox. Searching over strategies dramatically increases performance: The agent uses the score together with output logs and other information to hill climb towards a better score. We used a tree search (TS) strategy with an upper confidence bound (UCB) inspired by AlphaZero[13]. A critical difference from AlphaZero is that our problems don't allow exhaustive enumeration of all possible children of a node, so every node is a candidate for expansion. We therefore modify the UCB algorithm to count visits and compute mean values using the tree. However, when sampling a node to expand, we sample directly from the whole set instead of recursing from the root like AlphaZero.

We also note that the algorithm differs from traditional TS, in that the scoring of the nodes do not involve random rollouts (e.g. of a game) to estimate the value of a node. Yet there is still randomness for scoring each node, caused by the sampling of the LLM itself, which produces a distribution of different codes (scores) for each fixed prompt.

We use a PUCT tree search algorithm to explore the space of notebooks[12]. The PUCT (Predictor + Upper Confidence bound applied to Trees) algorithm is described in Algorithm 1. For tree T, and executed candidate u, we define the flat prior $P_T(u) = \frac{1}{|T|}$. To make it easier to tune the exploration constant c_{puct} across tasks, we convert task-specific scores $\text{TaskScore}(u)$ to rank scores $\text{RankScore}_T(u)$ in the PUCT formula. We define $\text{RankScore}_T(u) = \frac{\text{Rank}_T(u)-1}{|T|-1}$, when $|T| > 1$, and 1 otherwise, where $\text{Rank}_T(u)$ gives ascending-order ranks to the candidates.

Algorithm 1 UCB tree search (PUCT)

```
Input: GenerateAndExecute(), TaskScore() to define rank scores $\text{RankScore}_T(u)$, exploration constant $c_{puct}$, and a root node $r$.
1: $T \leftarrow \{r\}$                                   ▷ Initialize the tree with a root node.
2: $V(r) \leftarrow 1$
3: for all iterations do
4:     $N_{total} \leftarrow \sum_{u \in T} V(u)$          ▷ Get total visits across all nodes
5:     Select $u^* \leftarrow \text{argmax}_{u \in T} \left(\text{RankScore}_T(u) + c_{puct} P_T(u) \frac{\sqrt{N_{total}}}{1+V(u)}\right)$   ▷ Select node with highest PUCT score
6:     $u_c \leftarrow \text{GenerateAndExecute}(u^*)$      ▷ Expand the selected node and Execute
7:     $T \leftarrow T \cup \{u_c\}$
8:     $V(u_c) \leftarrow 1$
9:     for all ancestors $u_a$ of $u_c$ (excluding $u_c$) do   ▷ Backpropagate results
10:        $V(u_a) \leftarrow V(u_a) + 1$
11:    end for
12: end for
13: return $\text{argmax}_{u \in T} \text{TaskScore}(u)$      ▷ Best solution found
```

Adding Research Ideas to the Code Mutation System

When an expert solves difficult scientific problems, they often search for prior work for ideas. Prior work could be sourced from highly cited papers, specialized textbooks, or search engines. The search for prior work can also be powered by LLMs[35,36,75–78].

We emulate the expert behavior by injecting instructions for carrying out research ideas into the prompt of our code mutation system (Figure 1). We applied the research instruction injection for scRNA-seq batch integration, COVID prediction, segmenting remote sensing images, and whole-brain neural activity prediction. While the most successful outcomes used top methods from the literature, we also used two LLM driven search strategies: Deep Research from Gemini 2.5 Flash[35] and AI co-scientist[36].

For running these searches, we provided the tools with background information from the main problem description, and instructed the models to create distinct ideas (Supplementary Table 12). After manually filtering proposals and removing one proposed scRNA-seq batch integration method, we prompted Gemini to format the ideas into a structure consistent with our baseline method descriptions (Supplementary Table 13). Finally, we ran our method on these ideas to create empirical codes that could be scored.

Recombination Experiments

For both scRNA-seq batch integration problem and COVID-19 forecasting, we combined ideas from methods already generated using tree search. For the scRNA-seq batch integration problem, we used the first versions of our 11 baseline methods. For the COVID-19 prediction problem, we used the eight replications of models submitted to CovidHub. We first took the top-performing node from each tree search run seeded with one of these methods, based on its score on the validation set (for COVID-19 prediction, this included six weeks of reference dates from 2025-02-22 to 2025-03-29). Then, for every pair of these methods, we prompted Gemini 2.5 Flash to compare the two methods and explain the core technical similarities and differences between the two parent models using a consistent prompt (Supplementary Table 6). The explanatory response was then added to the the prompt, along with a statement instructing tree search to recombine the ideas by combining the best parts of both approaches (Supplementary Table 14). Subsequently, we ran our method to generate new hybrid strategies. This process yielded 55 recombined methods for the scRNA-seq batch integration problem, and 28 for the COVID-19 prediction problem (evaluated on the three-week holdout set 2025-04-05 to 2025-04-19, see Fig. 3d).

Gemini embeddings For each tree search implementation, we input the code snippets to the Gemini text embedding model[79], and the resulting 3,072-dimensional output vectors served as the semantic representations of their respective implementations.

scRNA-seq batch integration

For all scRNA-seq experiments, we ran tree search with 500 nodes. Each experiment took roughly seven hours to execute on our infrastructure.

Dataset We sourced a dataset from CZ CELLxGENE Discover[27] to use for hill climbing with tree search. To identify datasets distinct from the six OpenProblems.bio test datasets but that have similar characteristics, we filtered to datasets that contain only healthy human cells, with primary cell count $\geq$ 2,000, at least 10 unique cell types, at least seven unique donor ids (i.e. number of batches), and contain at least two unique assays that are also present in the OpenProblems.bio datasets. This filtering process identified 22 candidate datasets. After manually investigating the candidate datasets, we selected the dataset `364bd0c7-f7fd-48ed-99c1-ae26872b1042` version `ffdaa1f0-b1d1-4135-8774-9fed7bf039ba`[19].

Within the selected dataset, we applied quality control metrics and data processing steps identical to the processing performed on the OpenProblems.bio datasets[80,81], yielding a processed dataset with normalized expression values, highly variable genes, principal components, and k-nearest neighbors all computed. For computational efficiency, we randomly selected two disjoint subsets of $N = 20,000$ cells each, attempting to match `(batch, cell type)` distributions of the entire processed dataset. The "train" dataset was used for model training and selection of the highest-performing node in a single tree search. The "validation" dataset was used to select the best tree search for methods in which we ran multiple replicates of the same algorithm (Supplementary Fig. 1).

Evaluating scRNA-seq Batch Integration on the OpenProblems.bio Benchmark We downloaded the OpenProblems `v2.0.0` input and solution data from `s3://openproblems-data/resources/task_batch_integration/datasets/cellxgene_census/` and raw performance metrics from `s3://openproblems-data/resources/task_batch_integration/results/run_2025-01-23_18-03-16/score_uns.yaml`. We computed control-scaled metric results identically to the published OpenProblems results. Briefly, for each `(dataset, metric)`, lower and upper bounds on raw scores are defined as the minimum and maximum values achieved by the seven "control" methods. Raw values were linearly scaled between those extrema and clamped to be in $[0, 1]$. Overall score was computed as the arithmetic mean over all 78 measurements (13 metrics computed for each of 6 datasets) with NaN values replaced by 0 (i.e., failure to compute a metric causes it to be considered the worst possible score).

Replication of Existing Methods for Batch Integration The OpenProblems.bio benchmark profiles the performance of several state-of-the-art existing methods. As of July 11, 2025 there were 19 different methods. Three methods have implementations in both R and Python: `LIGER` and `pyliger`, `Harmony` and `Harmonypy`, and `batchelor mnnCorrect` and `mnnpy`. After grouping reimplementations of the same method, there are 16 separate research ideas. From this list, we excluded all six foundation model methods (`UCE`, `SCimilarity`, `scGPT (zero shot)`, `scGPT (fine-tuned)`, `Geneformer`, and `scPRINT`) because they perform very poorly on the benchmark and use a much larger training set. For example, only a single foundation model (`UCE`) performs better than the negative control of "No integration" which simply performs PCA on the dataset. We further excluded `scANVI`, which is a modification of `scVI` that is trained using cell type information. Since cell type information is used to define the metrics, this represents data leakage and consequently we consider `scANVI` a control method. This resulted in nine existing different research methods to optimize with tree search.

For each of the nine existing methods, we obtained the manuscript PDF corresponding to the method. To obtain a short method description from the manuscript, we used Gemini 2.5 Pro Thinking to summarize the paper (prompt in Supplementary Table 15, example output in Supplementary Table 16). For `batchelor fastMNN`, which is a faster implementation of `batchelor mnnCorrect`, there is no separate publication and thus we provided the paper PDF of `batchelor mnnCorrect` as well as the docstring corresponding to `batchelor fastMNN` from `https://rdrr.io/github/LTLA/batchelor/man/fastMNN.html` (Details section) with a slightly adjusted prompt. Finally, the method summary is added to the tree search notebook, and is used to come up with better code solutions given the method summary.

For each of the nine methods, we ran three replicates of tree search. For Fig. 2, we selected the replicate that had the best performance based on the validation set score. We show the performance of all replicates in Supplementary Fig. 3.

Hyperparameters To determine optimal hyperparameters for each base method, we employed Optuna, an automated hyperparameter optimization framework[82]. Search spaces were defined across integer, float, and categorical parameter types by experts. The optimization process ran for a total of five times the number of parameters. In each trial, a model was trained using a sampled parameter set and evaluated based on a performance metric that Optuna's Tree-structured Parzen Estimator (TPE) sampler aimed to maximize. All hyperparameter optimization was conducted solely on the training dataset. The best identified hyperparameter set was then utilized to train the final base methods and evaluate them on the held-out OpenProblems dataset.

COVID-19 prediction

Dataset Our primary data source was historical confirmed COVID-19 hospital admissions, which corresponds to the target variable specified by CovidHub. These data are published weekly by the CDC within the National Healthcare Safety Network (NHSN) Hospital Respiratory Data (HRD) dataset[83]. Preprocessing was kept minimal–missing values in the dataset were replaced by zeros to enable tree search to find executable code with the criterion score (WIS). The only additional data source used to augment the target for our model was static jurisdiction-specific population values from the CovidHub GitHub Repository[21]. For comparing model performance in Fig. 3c, we use all of the models submitted to Forecast Hub which make predictions at a state by state level and have forecasts for at least 75 percent of the season and time horizons. We ran tree search with 2000 nodes for each reported run.

Replication of existing COVID-19 prediction models We selected eight models for replication from those that had submitted to CovidHub based on the following inclusion criteria: (1) The method must be reproducible solely using historical COVID-19 hospitalization data, without reliance on external predictor variables, (2) The model submission must include predictions across all specified time horizons, and (3) Model submissions must be available for over three months (12 weeks) to enable meaningful comparison. Three models were excluded for failing these criteria: two were ensembles of external forecasts, and one relied entirely on additional data. An additional five models were excluded because they did not provide predictions for all forecast horizons. These five models originated from the same forecasting team. As all our analysis involves aggregating model performance across horizons, we have excluded these five models from all comparisons. Overall this gave a selection of eight models for replication.

To instruct the search algorithm, we provided the method descriptions from the original authors' official submission metadata. For example, the metadata for the `UMASS-arc6-pooled` model states: "*AR(6) model after fourth root data transform. AR coefficients are shared across all locations. A separate variance parameter is estimated for each location.*" We integrated these concise descriptions directly into the tree search prompt as part of the model directions, transforming them into instructions by prepending 'Use a/an' (see Methods, Supplementary Table 8).

GIFT-Eval Benchmark

We applied our tree search methodology to the General Time Series Forecasting Model Evaluation (GIFT-Eval) benchmark[24]. The search begins from a root node defined by an initial code template and proceeds via hill climbing, where new candidate solutions are generated and evaluated against the GIFT-Eval validation folds. At the end of a tree search, we evaluated the solution on the held-out test set using MASE point forecast as the scoring metric. Our results are based on a 5/18/2025 snapshot of the dataset, official leaderboard and scoring, all of which have been updated since. See Supplementary Table 10 for a complete snapshot of the leaderboard.

We adhered to the benchmark's framework, utilizing the official dataset source from Hugging Face, its pre-defined training, validation, and test splits, as well as the scoring and evaluation code commonly used in the existing submission notebooks.

Per-dataset Solution We conducted separate tree searches for 92 of the 97 GIFT-Eval datasets, excluding the five largest due to computational constraints; for these, the naive baseline score was used in order to produce the aggregated leaderboard score. For each dataset, we used a search of 300 nodes, with the agent permitted to use a broad suite of machine learning libraries, including `scikit-learn`, `XGBoost`, and `statsmodels`. Supplementary Fig. 17 shows an analysis of the types of models used across the 92 different solutions.

Unified Solution Here, we created a single, unified forecasting library that could generalize across all 97 datasets. We used a tree search of over 1,000 nodes, guided by the geometric mean of the normalized MASE scores across all datasets, providing a single objective function to optimize. To force the model to reason from first principles, its access was restricted to basic libraries (`numpy`, `pandas`, and `holidays`).

The resulting solution consists of two components: a single forecasting library and a list of eight preset configurations. For each dataset, the best-performing configuration is identified on the validation set. This selected configuration is then used with the unified library to produce the final forecast on the test set, allowing the model to adapt its strategy without seeing test data.

The final solution was developed iteratively. An initial search yielded a base model with a MASE of 0.82. A key breakthrough occurred in a subsequent run when the search space was expanded to ten configurations and the agent was advised to use the `holidays` library, which improved the MASE to 0.77 (Supplementary Fig. 18). A final 500 node refinement run pruned the configurations to an optimized set of eight, achieving the final MASE of 0.734.

The final solution sequentially models and removes fundamental components of the series, with the final forecast being the sum of the individual component forecasts. This approach allows the model to be highly configurable while systematically accounting for different sources of variation in the data. This process is outlined with the following steps:

1. **Preprocessing:** The input series first undergoes basic cleaning, including median imputation for any missing values. An optional log-transform (`log1p`) can be applied to stabilize variance in series with exponential growth patterns.
2. **Base/Level Component:** A base level is established using simple but robust methods like a seasonal naive forecast or a rolling median of recent data points. This component captures the basic magnitude of the series.
3. **Trend Component:** The residuals from the base component are then modeled to capture linear or polynomial trends. This step includes a `damping_factor` to prevent unrealistic long-term extrapolation by gradually flattening the trend.
4. **Seasonality Component:** The residuals from the trend component are analyzed to model cyclical patterns (e.g., weekly, yearly). The model identifies the cycle length and forecasts seasonality by averaging values at the same point in the cycle (e.g., the average value for all Mondays).
5. **Datetime and Holiday Features:** To capture special events and non-seasonal cycles, features are extracted from the timestamp (e.g., `dayofweek`, `is_holiday_flag`). The model calculates the median effect of each feature category from the remaining residuals and adds it to the forecast.
6. **Residual Correction:** As a final step, a correction is made by modeling the median of the most recent unexplained errors. This autoregressive-like step helps correct for short-term biases in

the model. A decay_factor fades its impact over the forecast horizon.

To apply the unified solution to a new dataset, one would first split the historical data into training and validation sets. Using the library's adaptive configuration system, one can then find a suitable forecasting strategy by evaluating the eight preset configurations on the validation data to select the best-performing one. This provides a strong, data-driven starting point that can be used directly. For more specialized applications, one can also create a custom configuration, allowing for manual refinement of the model's components and making the library both powerful out-of-the-box and flexible enough for expert tuning.

Difficult Integrals

We carried out a tree search over 1000 nodes, using a list of integrals in Supplementary Fig. 21.

To build these lists of integrals, we started with a long list of integrals in LaTeX form from Gradshteyn and Ryzhik[60]. We converted both the question and solution into a python expression using SymPy[84]. Most expressions included free parameters, often with value constraints. To enable numeric evaluations, we generated random values for all parameters consistent with the constraints.

Once an integral and its answer were in the form of SymPy expression objects, we evaluated answers numerically by substituting our chosen parameter values using `sympy.Expr.subs()` and evaluating via `sympy.evalf()`. We build integrand functions suitable for `scipy.integrate.quad()` via `sympy.lambdify` for efficient evaluation. We compared each numerical answer to the number returned by `scipy.integrate.quad()` and discarded cases where the numbers agreed within the latter's error estimate. We also discarded cases where that error estimate was greater than 2% of the numbers' magnitude.

All conversions from LaTeX to SymPy and all constrained parameter generations were performed by Gemini using specialized prompts. The resulting SymPy expressions and parameter values were examined manually for correctness. These manual steps were the limiting factor on the scale of our dataset.

The scoring function we used during training used the absolute fractional error (discrepancy between the generated solution's number and the answer's number) via a logarithm to prevent outliers from dominating the result.

$$\text{score} = -\log\left(1 + \left|\frac{\text{response} - \text{answer}}{\text{answer}}\right|\right) \tag{1}$$

References

[1] Fortin, J. A., Cardille, J. A. & Perez, E. Multi-sensor detection of forest-cover change across 45 years in Mato Grosso, Brazil. *Remote Sens. Environ.* **238**, 111266 (2020).

[2] Hohenberg, P. & Kohn, W. Inhomogeneous electron gas. *Phys. Rev.* **136**, B864 (1964).

[3] Kohn, W. & Sham, L. J. Self-consistent equations including exchange and correlation effects. *Phys. Rev.* **140**, A1133 (1965).

[4] Warshel, A. & Levitt, M. Theoretical studies of enzymic reactions: dielectric, electrostatic and steric stabilization of the carbonium ion in the reaction of lysozyme. *J. Mol. Biol.* **103**, 227–249 (1976).

[5] Jumper, J. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596**, 583–589 (2021).

[6] Baek, M. *et al.* Accurate prediction of protein structures and interactions using a three-track neural network. *Science* **373**, 871–876 (2021).

[7] Hourdin, F. *et al.* The art and science of climate model tuning. *Bull. Am. Meteorol. Soc.* **98**, 589–602 (2017).

[8] Anderson Jr., J. Basic philosophy of CFD. In *Computational Fluid Dynamics*, 3–14 (Springer, 2009).

[9] Silver, N. *The signal and the noise: why so many predictions fail-but some don't* (Penguin, 2012).

[10] Farmer, J. D. *Making sense of chaos: a better economics for a better world* (Yale Univ. Press, 2024).

[11] Bernanke, B. & Blanchard, O. What caused the US pandemic-era inflation? *Am. Econ. J. Macroecon.* **17**, 1–35 (2025).

[12] Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).

[13] Silver, D. *et al.* Mastering the game of Go without human knowledge. *Nature* **550**, 354–359 (2017).

[14] Jiang, Z. *et al.* AIDE: AI-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138* (2025).

[15] Novikov, A. *et al.* AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131* (2025).

[16] Romera-Paredes, B. *et al.* Mathematical discoveries from program search with large language models. *Nature* **625**, 468–475 (2024).

[17] Wu, X., Wu, S.-h., Wu, J., Feng, L. & Tan, K. C. Evolutionary computation in the era of large language model: survey and roadmap. *IEEE Trans. Evol. Comput.* (2024).

[18] Hu, S., Lu, C. & Clune, J. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435* (2024).

[19] Xu, C. *et al.* Automatic cell-type harmonization and integration across Human Cell Atlas datasets. *Cell* **186**, 5876–5891.e20 (2023).

[20] Regev, A. *et al.* The Human Cell Atlas. *eLife* **6**, e27041 (2017).

[21] Centers for Disease Control and Prevention. COVID-19 forecast hub (2025). URL https://github.com/cdcgov/covid19-forecast-hub?tab=readme-ov-file.

[22] Shao, Z., Yang, K. & Zhou, W. Performance evaluation of single-label and multi-label remote sensing image retrieval using a dense labeling dataset. *Remote Sens.* **10**, 964 (2018).

[23] Lueckmann, J.-M. *et al.* ZAPBench: a benchmark for whole-brain activity prediction in zebrafish. *arXiv preprint arXiv:2503.02618* (2025).

[24] Aksu, T. *et al.* GIFT-Eval: a benchmark for general time series forecasting model evaluation. *arXiv preprint arXiv:2410.10393* (2024). URL https://huggingface.co/spaces/Salesforce/GIFT-Eval.

[25] Jovic, D. *et al.* Single-cell RNA sequencing technologies and applications: a brief overview. *Clin. and Transl. Med.* **12**, e694 (2022).

[26] Svensson, V., Vento-Tormo, R. & Teichmann, S. A. Exponential scaling of single-cell RNA-seq in the past decade. *Nat. Protoc.* **13**, 599–604 (2018).

[27] CZI Cell Science Program *et al.* CZ CELLxGENE Discover: a single-cell data platform for scalable exploration, analysis and modeling of aggregated data. *Nucleic Acids Res.* **53**, D886–D900 (2025).

[28] Zhang, J. *et al.* Tahoe-100M: a giga-scale single-cell perturbation atlas for context-dependent gene function and cellular modeling. *bioRxiv* 2025–02 (2025).

[29] Stuart, T. & Satija, R. Integrative single-cell analysis. *Nat. Rev. Genet.* **20**, 257–272 (2019).

[30] Zappia, L., Phipson, B. & Oshlack, A. Exploring the single-cell RNA-seq analysis landscape with the scRNA-tools database. *PLoS Comput. Biol.* **14**, e1006245 (2018).

[31] Tran, H. T. N. *et al.* A benchmark of batch-effect correction methods for single-cell RNA sequencing data. *Genome Biol.* **21**, 1–32 (2020).

[32] Chazarra-Gil, R., van Dongen, S., Kiselev, V. Y. & Hemberg, M. Flexible comparison of batch correction methods for single-cell RNA-seq using BatchBench. *Nucleic Acids Res.* **49**, e42 (2021).

[33] Luecken, M. D. *et al.* Benchmarking atlas-level data integration in single-cell genomics. *Nat. Methods* **19**, 41–50 (2022).

[34] Luecken, M. D. *et al.* Defining and benchmarking open problems in single-cell analysis. *Nat. Biotechnol.* **43**, 1035–1040 (2025).

[35] Google. Gemini Deep Research (2025). URL https://gemini.google/overview/deep-research/?hl=en.

[36] Gottweis, J. *et al.* Towards an AI co-scientist. *arXiv preprint arXiv:2502.18864* (2025).

[37] Johnson, W. E., Li, C. & Rabinovic, A. Adjusting batch effects in microarray expression data using empirical Bayes methods. *Biostatistics* **8**, 118–127 (2007).

[38] Polański, K. *et al.* BBKNN: fast batch alignment of single cell transcriptomes. *Bioinformatics* **36**, 964–965 (2019).

[39] Chandrashekar, A. *et al.* TabVI: leveraging lightweight transformer architectures to learn biologically meaningful cellular representations. *bioRxiv* 2025–02 (2025).

[40] Yang, Y. & Newsam, S. Bag-of-visual-words and spatial extensions for land-use classification. In *Proc. 18th SIGSPATIAL Int. Conf. on Adv. in Geogr. Inf. Syst.*, 270–279 (Association for Computing Machinery, 2010).

[41] Russakovsky, O. *et al.* ImageNet large scale visual recognition challenge. *Int. J. Comput. Vis.* **115**, 211–252 (2015).

[42] Krizhevsky, A., Sutskever, I. & Hinton, G. E. ImageNet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **25** (2012).

[43] Zhong, B., Du, J., Liu, M., Yang, A. & Wu, J. Region-enhancing network for semantic segmentation of remote-sensing imagery. *Sensors* **21** (2021).

[44] Zhang, Z., Liu, B. & Li, Y. FURSformer: semantic segmentation network for remote sensing images with fused heterogeneous features. *Electronics* **12** (2023).

[45] Atiampo, A. K. & Diédié, G. H. F. New fusion approach of spatial and channel attention for semantic segmentation of very high spatial resolution remote sensing images. *Open J. Appl. Sci.* **14**, 288–319 (2024).

[46] Sun, Y., Bi, F., Gao, Y., Chen, L. & Feng, S. A multi-attention UNet for semantic segmentation in remote sensing images. *Symmetry* **14**, 906 (2022).

[47] Elgamily, K. M., Mohamed, M. A., Abou-Taleb, A. M. & Ata, M. M. A novel W13 deep CNN structure for improved semantic segmentation of multiple objects in remote sensing imagery. *Neural Comput. Appl.* **37**, 5397–5427 (2025).

[48] Immer, A. *et al.* Forecasting whole-brain neuronal activity from volumetric video. *arXiv preprint arXiv:2503.00073* (2025).

[49] Zeng, A., Chen, M., Zhang, L. & Xu, Q. Are transformers effective for time series forecasting? In *Proc AAAI Conf. Artif. Intell.*, vol. 37, 11121–11128 (2023).

[50] Das, A. *et al.* Long-term forecasting with TiDE: Time-series Dense Encoder. *Trans. Mach. Learn. Res.* (2023).

[51] Chen, S.-A., Li, C.-L., Yoder, N., Arik, S. O. & Pfister, T. TSMixer: An All-MLP architecture for time series forecasting. *Trans. Mach. Learn. Res.* (2023).

[52] Perez, E., Strub, F., De Vries, H., Dumoulin, V. & Courville, A. FiLM: Visual reasoning with a general conditioning layer. In *Proc AAAI Conf. Artif. Intell.*, vol. 32 (2018).

[53] Deistler, M. *et al.* Differentiable simulation enables large-scale training of detailed biophysical models of neural dynamics. *bioRxiv* 2024–08 (2024).

[54] Hoo, S. B., Müller, S., Salinas, D. & Hutter, F. From tables to time: how TabPFN-v2 outperforms specialized time series forecasting models. *arXiv preprint arXiv:2501.02945* (2025).

[55] Liu, Y. *et al.* Sundial: A family of highly capable time series foundation models. *arXiv preprint arXiv:2502.00816* (2025).

[56] Ansari, A. F. *et al.* Chronos: learning the language of time series. *Trans. Mach. Learn. Res.* (2024).

[57] Oreshkin, B. N., Carpov, D., Chapados, N. & Bengio, Y. N-BEATS: neural basis expansion analysis for interpretable time series forecasting. *arXiv preprint arXiv:1905.10437* (2019).

[58] Ho, S. L. & Xie, M. The use of ARIMA models for reliability forecasting and analysis. *Comput. Ind. Eng.* **35**, 213–216 (1998).

[59] Piessens, R., de Doncker-Kapenga, E., Überhuber, C. W. & Kahaner, D. *QUADPACK: a subroutine package for automatic integration* (Springer-Verlag, 1983).

[60] Gradshteyn, I. & Ryzhik, I. *Table of integrals, series, and products, 8th edn* (Academic Press, 1994).

[61] Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Stat. Comput.* **4**, 87–112 (1994).

[62] Mernik, M., Heering, J. & Sloane, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* **37**, 316–344 (2005).

[63] Czarnecki, K. Generative programming: Methods, techniques, and applications tutorial abstract. In *International Conference on Software Reuse*, 351–352 (Springer, 2002).

[64] Chen, M. *et al.* Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[65] Li, Y. *et al.* Competition-level code generation with AlphaCode. *Science* **378**, 1092–1097 (2022).

[66] Hutter, F., Kotthoff, L. & Vanschoren, J. *Automated machine learning: methods, systems, challenges* (Springer Nature, 2019).

[67] Merchant, A. *et al.* Scaling deep learning for materials discovery. *Nature* **624**, 80–85 (2023).

[68] Xiao, Y. *et al.* CellAgent: An LLM-driven multi-agent framework for automated single-cell data analysis. *arXiv preprint arXiv:2407.09811* (2024).

[69] Zhang, H. *et al.* CompBioAgent: An LLM-powered agent for single-cell RNA-seq data exploration. *bioRxiv* 2025–03 (2025).

[70] Zhou, J. *et al.* An AI agent for fully automated multi-omic analyses. *Adv. Sci.* **11**, 2407094 (2024).

[71] Xin, Q. *et al.* BioInformatics Agent (BIA): unleashing the power of large language models to reshape bioinformatics workflow. *bioRxiv* 2024–05 (2024).

[72] Alber, S. *et al.* CellVoyager: AI compbio agent generates new insights by autonomously analyzing biological data. *bioRxiv* 2025–06 (2025).

[73] Baek, J., Jauhar, S. K., Cucerzan, S. & Hwang, S. J. ResearchAgent: iterative research idea generation over scientific literature with large language models. *arXiv preprint arXiv:2404.07738* (2024).

[74] Lu, C. *et al.* The AI Scientist: towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292* (2024).

[75] Du, M., Xu, B., Zhu, C., Wang, X. & Mao, Z. DeepResearch Bench: a comprehensive benchmark for deep research agents. *arXiv preprint arXiv:2506.11763* (2025).

[76] Perplexity. Perplexity Deep Research (2025). URL https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research.

[77] Coelho, J. *et al.* DeepResearchGym: A free, transparent, and reproducible evaluation sandbox for deep research. *arXiv preprint arXiv:2505.19253* (2025).

[78] Xu, R. & Peng, J. A comprehensive survey of deep research: Systems, methodologies, and applications. *arXiv preprint arXiv:2506.12594* (2025).

[79] Lee, J. *et al.* Gemini Embedding: Generalizable embeddings from Gemini. *arXiv preprint arXiv:2503.07891* (2025).

[80] Gigante, S., Cannoodt, R. *et al.* `openproblems` (2025). URL https://github.com/openproblems-bio/openproblems.

[81] Cannoodt, R., Zappia, L., Burkhardt, D. *et al.* `task_batch_integration` (2025). URL https://github.com/openproblems-bio/task_batch_integration.

[82] Akiba, T., Sano, S., Yanase, T., Ohta, T. & Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.* (2019).

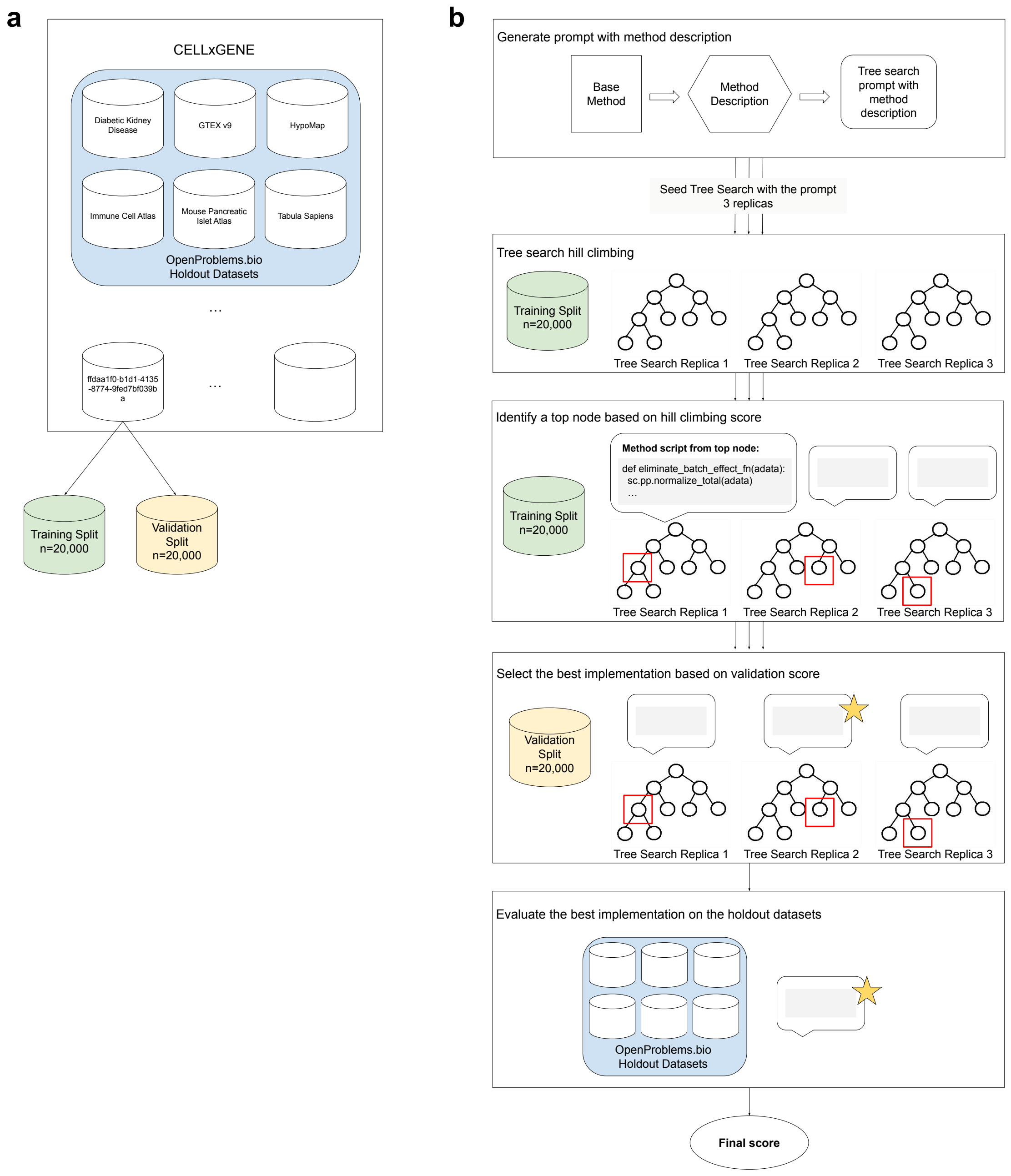
[83] Centers for Disease Control and Prevention. Weekly Hospital Respiratory Data (HRD) Metrics by Jurisdiction (2024). URL https://data.cdc.gov/Public-Health-Surveillance/Weekly-Hospital-Respiratory-Data-HRD-Metrics-by-Ju/mpgq-jmmr. Dataset ID: mpgq-jmmr. Last updated: June 14, 2024.

[84] Meurer, A. *et al.* SymPy: symbolic computing in Python. *PeerJ Comput. Sci.* **3**, e103 (2017).

[85] McInnes, L., Healy, J. & Melville, J. UMAP: uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

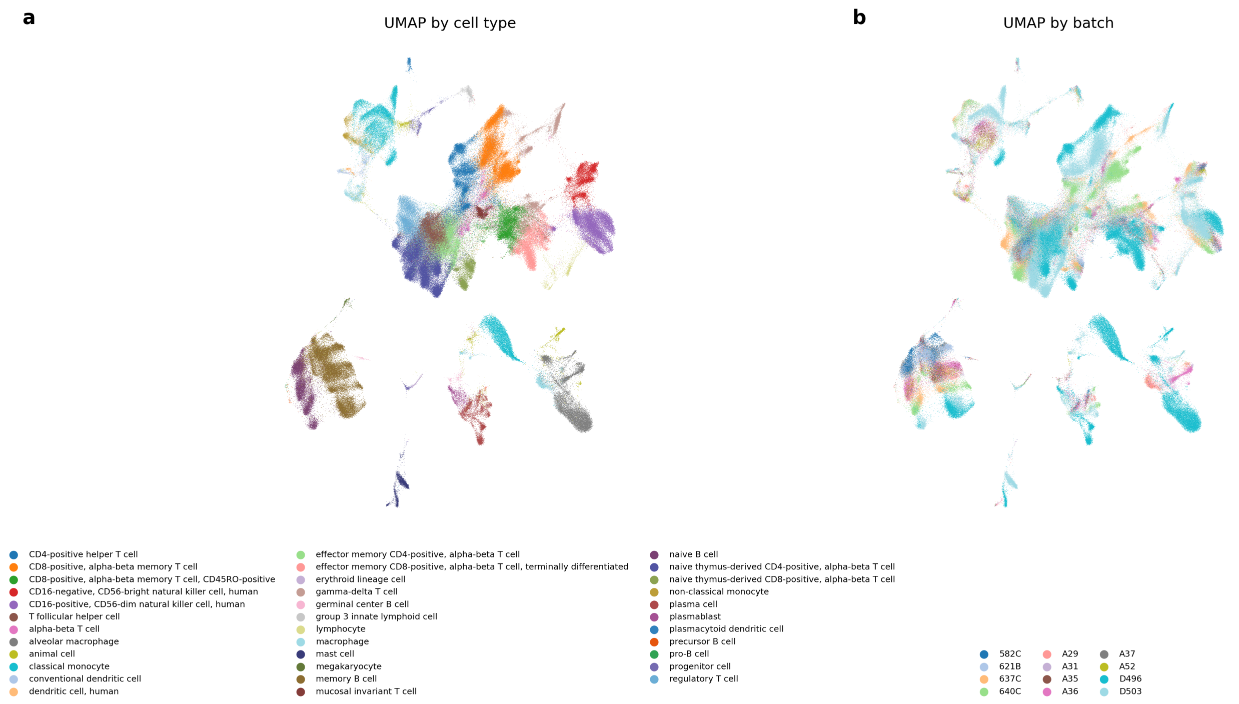
[86] Polański, K. *et al.* `bbknn` (2018). URL https://github.com/Teichlab/bbknn.

Supplementary Figures

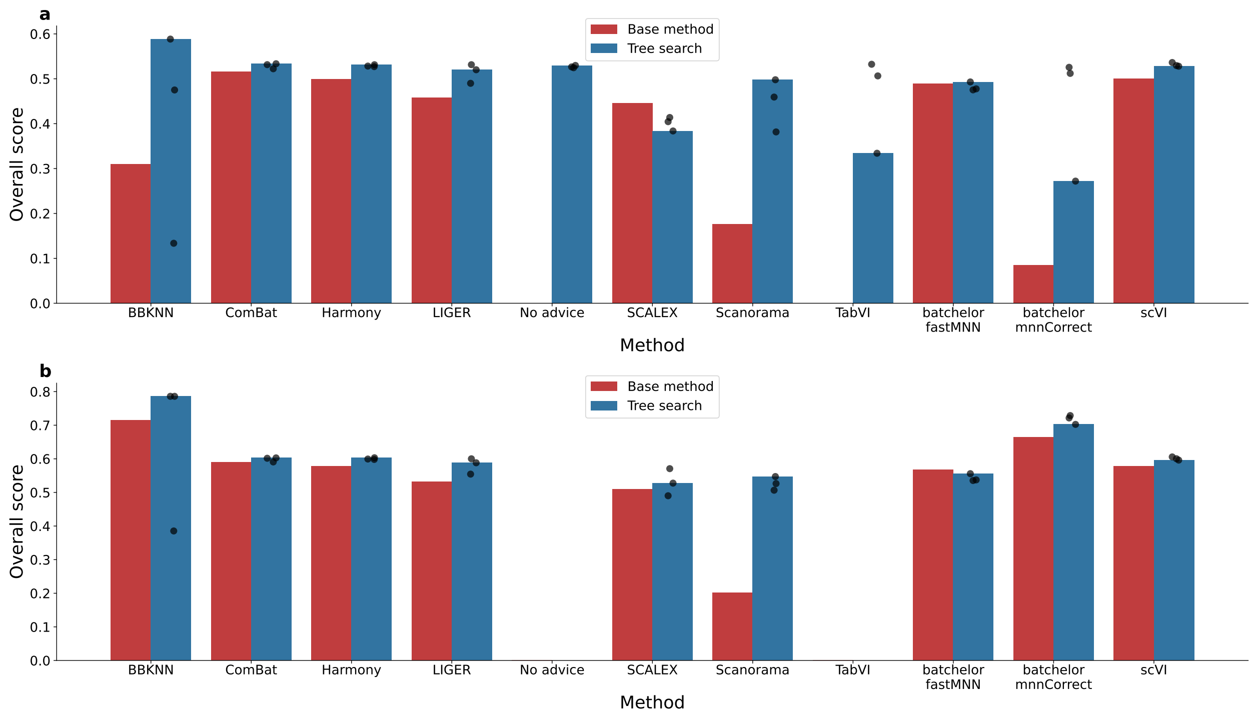

a
CELLxGENE
Diabetic Kidney Disease
GTEX v9
HypoMap
Immune Cell Atlas
Mouse Pancreatic Islet Atlas
Tabula Sapiens
OpenProblems.bio Holdout Datasets
…
ffdaa1f0-b1d1-4135-8774-9fed7bf039ba
…
Training Split n=20,000
Validation Split n=20,000
b
Generate prompt with method description
Base Method
Method Description
Tree search prompt with method description
Seed Tree Search with the prompt 3 replicas
Tree search hill climbing
Training Split n=20,000
Tree Search Replica 1
Tree Search Replica 2
Tree Search Replica 3
Identify a top node based on hill climbing score
Method script from top node:
Training Split n=20,000
Tree Search Replica 1
Tree Search Replica 2
Tree Search Replica 3
Select the best implementation based on validation score
Validation Split n=20,000
Tree Search Replica 1
Tree Search Replica 2
Tree Search Replica 3
Evaluate the best implementation on the holdout datasets
OpenProblems.bio Holdout Datasets
Final score

```
def eliminate_batch_effect_fn(adata):
  sc.pp.normalize_total(adata)
  …
```

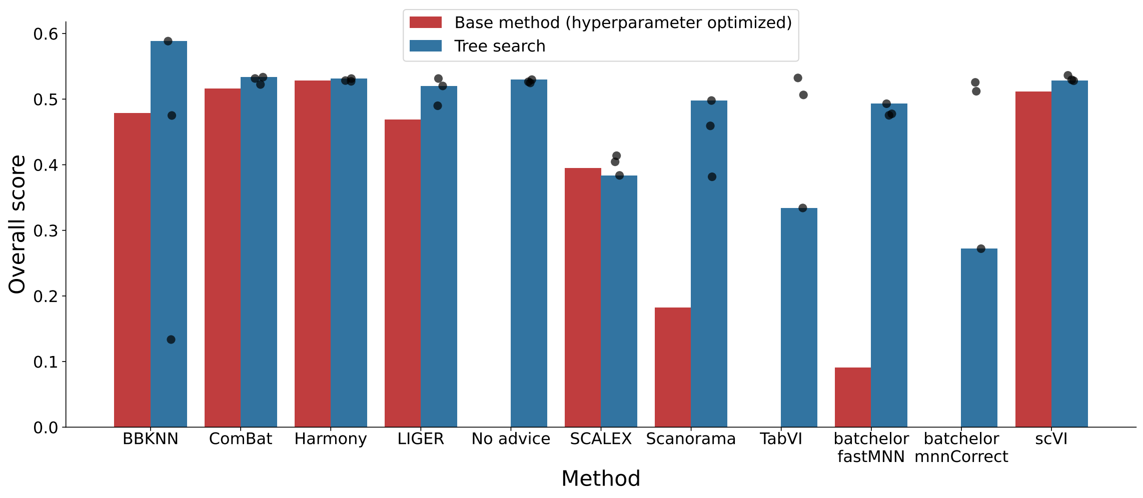
Supplementary Fig. 1 | Experimental design for single-cell batch integration. a, We sourced our tree search development dataset from CELLxGENE. After filtering and manually selecting the dataset `364bd0c7-f7fd-48ed-99c1-ae26872b1042` version `ffdaa1f0-b1d1-4135-8774-9fed7bf039ba` (see Methods), which has a similar profile to the six datasets used in the OpenProblems.bio Batch Integration benchmark (distinct datasets also in CELLxGENE), we sampled 20,000 cells for the training split and 20,000 for the validation split. **b,** For each of the 11 base methods, we generated a detailed method description and inserted it into a prompt to initialize the tree search. We ran three independent tree search replicas per method, using the training split for hill climbing. From each tree, we selected the top-performing node based on its training score. We then evaluated each top node's script on the validation split and selected the best one based on validation performance. The best implementation per method was finally evaluated on the OpenProblems.bio holdout datasets, and the corresponding scores are reported as final results.



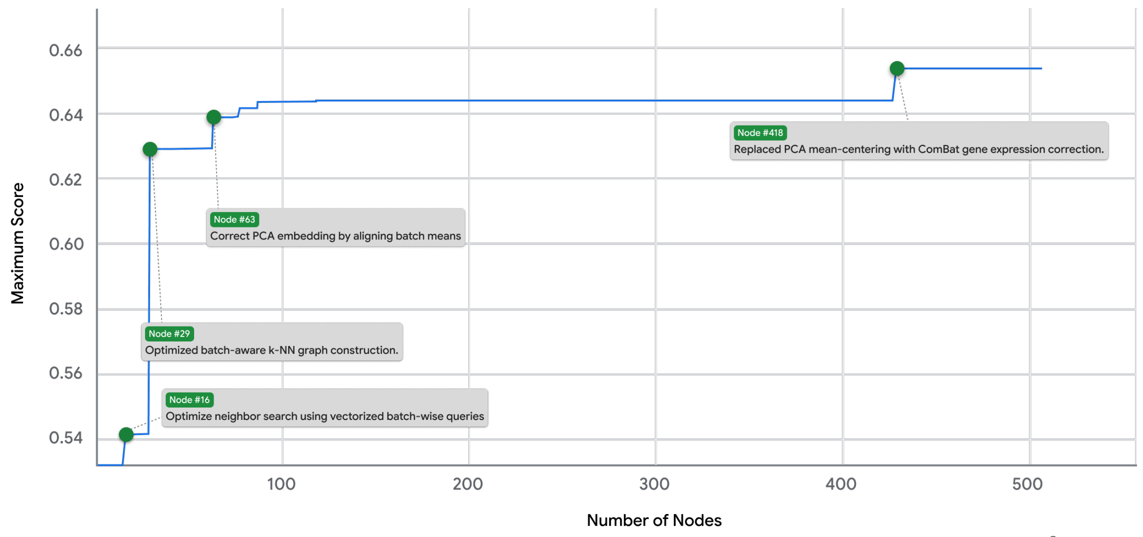
Supplementary Fig. 2 | Uniform Manifold Approximation and Projection[85] of `BBKNN (TS)` on the Immune Cell Atlas dataset. a, The UMAP projection colored by cell type shows cell-type-specific clusters. **b,** The UMAP projection colored by data batch shows good batch mixing across the dataset.

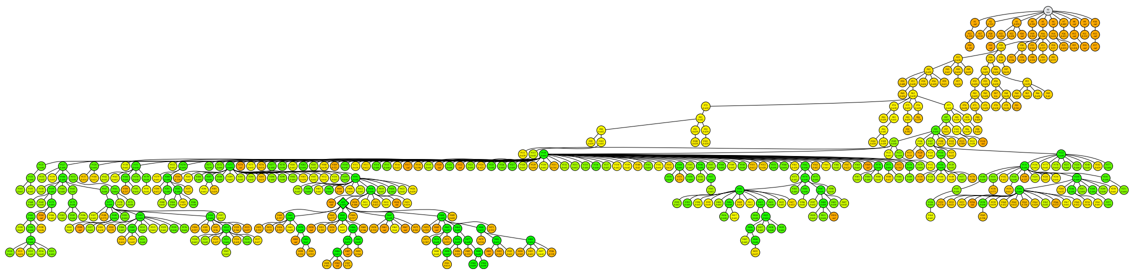


Supplementary Fig. 3 | Relative performance of base methods and our method replicates. a, Overall scores on the holdout OpenProblems datasets for all replicates of methods evaluated in Fig. 2. For tree search implementations, three replicates of the full process were performed. Dots indicate the overall score of the replicate on the holdout OpenProblems datasets. The bar shows the performance of the replicate with highest performance in the validation dataset (identical values to those shown in Fig. 2). The lowest performing tree search replicates for BBKNN, Scanorama, and TabVI only successfully computed 30, 57, and 45 of the 78 metrics, respectively. We note that failures due to out of memory or compute time issues were not explicitly selected against in our algorithm since all optimization was performed on datasets of only 20k cells. **b,** Average scores for each method when restricting to only `(method, dataset, metric)` combinations that have non-NaN values for the base method and all three tree search replicates. `No advice` and `TabVI` are absent since they have no base method comparator.

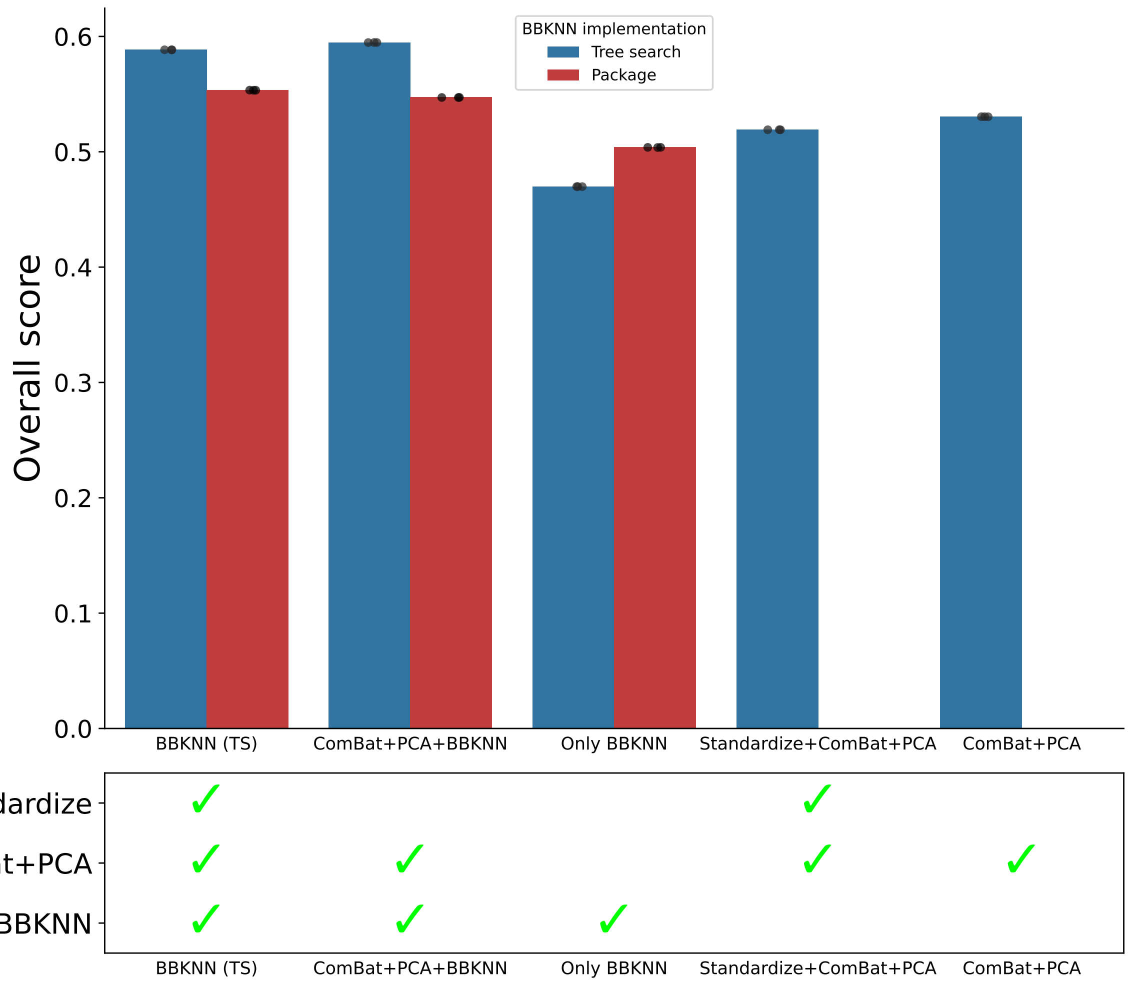


Supplementary Fig. 4 | Relative performance of base methods with optimized hyperparameters and tree search replicates. Overall scores on the holdout OpenProblems datasets for all replicates of methods evaluated in Fig. 2. Hyperparameters for the base methods were optimized using the training dataset. For tree search implementations, three replicates of the full process were performed. Dots indicate the overall score of the replicate on the holdout OpenProblems datasets. The bar shows the performance of the replicate with highest performance in the validation dataset (identical values to those shown in Fig. 2). The `No advice` and `TabVI` methods have no base method code available. The `batchelor mnnCorrect` hyperparameter-optimized base method code failed to compute embeddings on every OpenProblems dataset owing to out-of-memory errors.

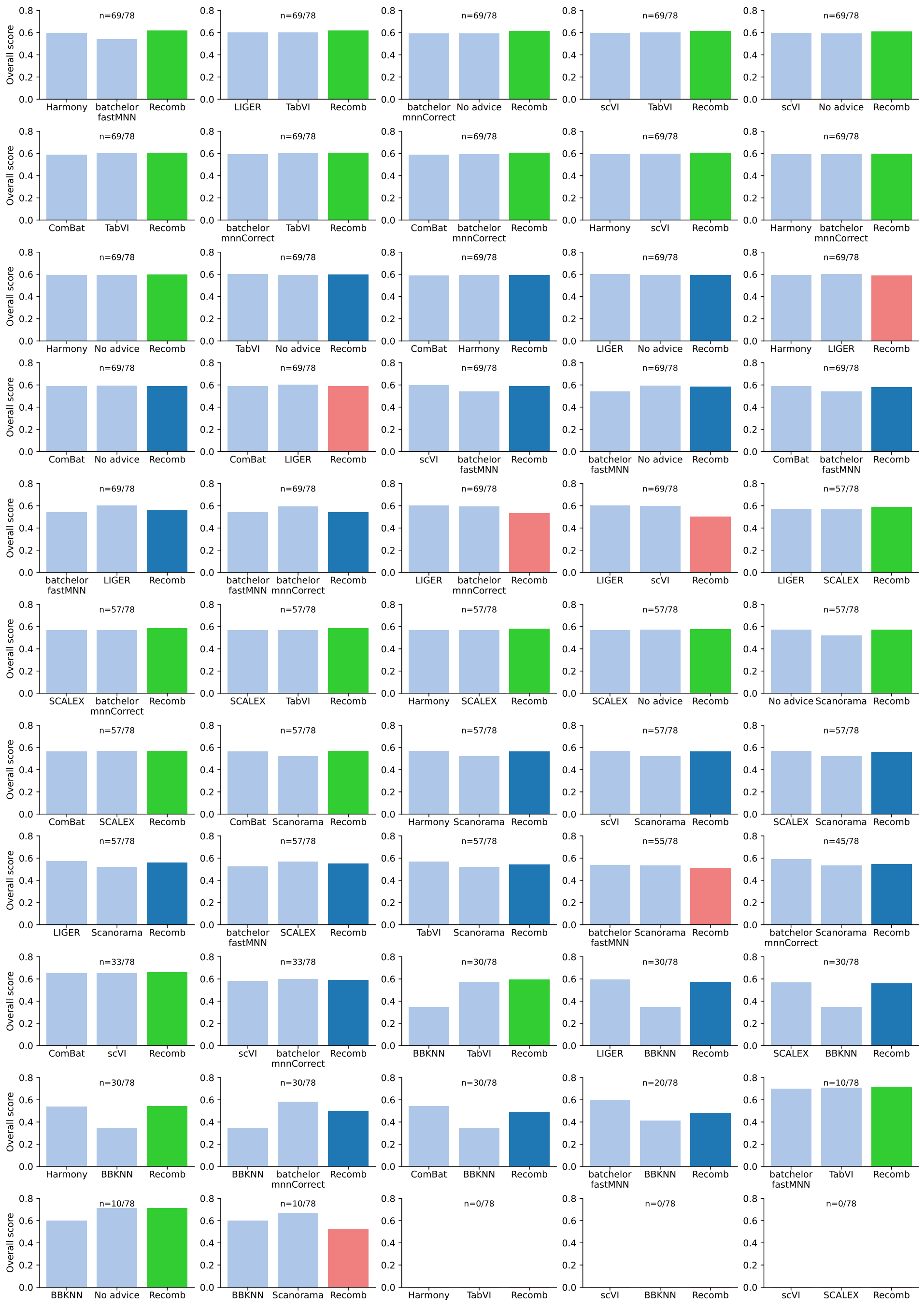




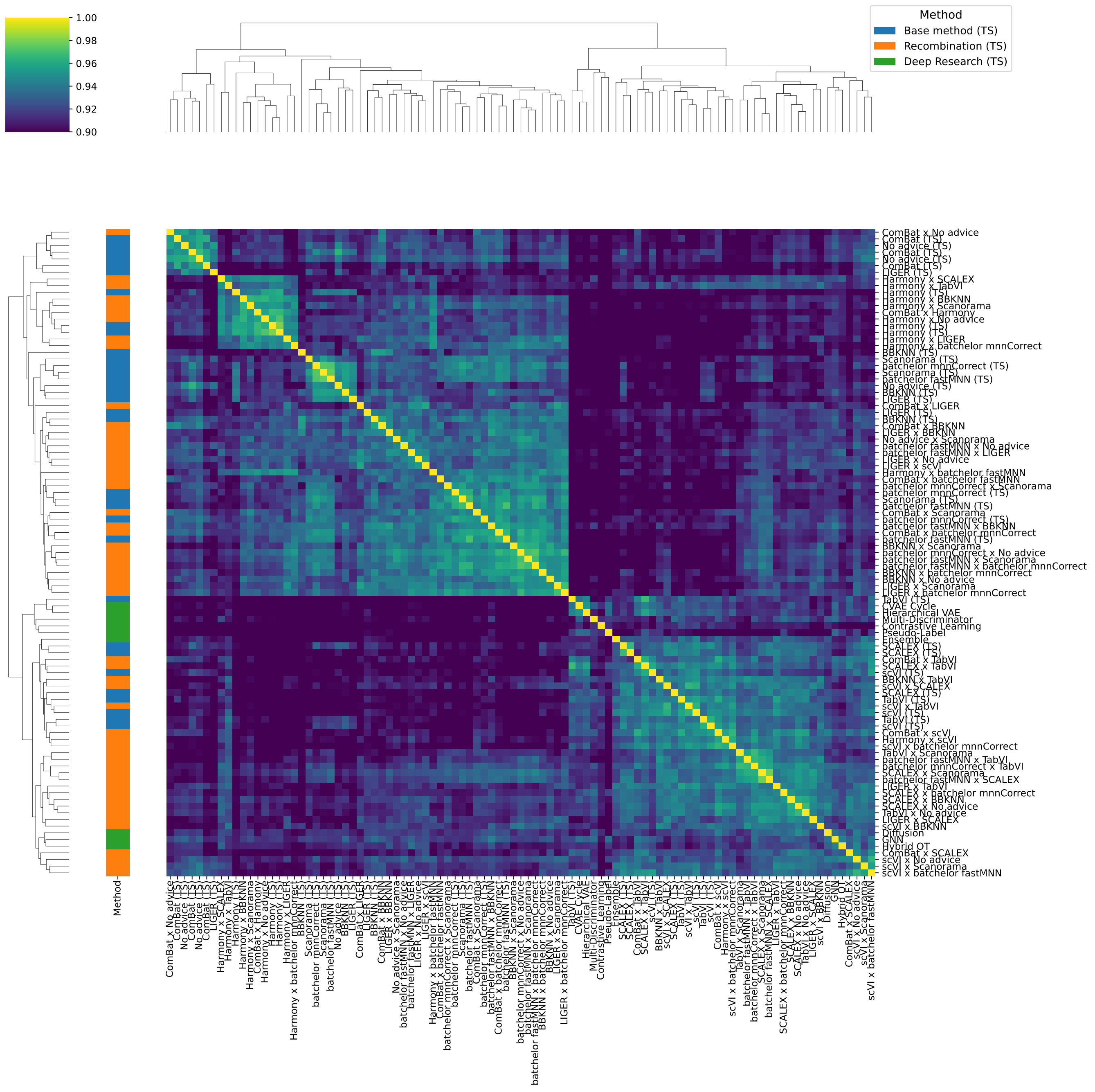
Supplementary Fig. 5 | *Top Figure* Breakthrough plot for the BBKNN (TS) tree search, showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.



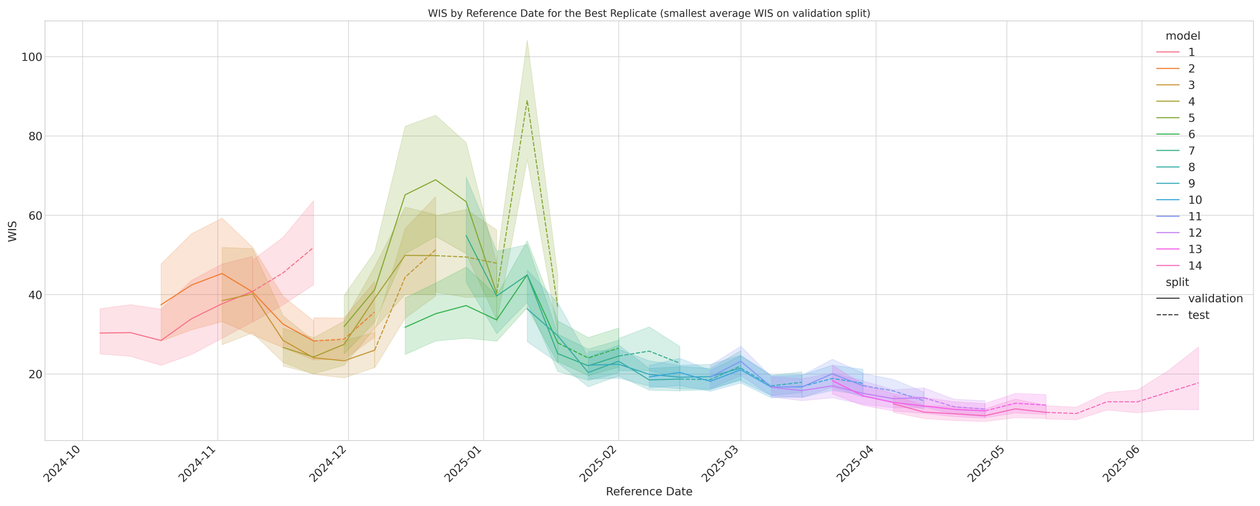
Supplementary Fig. 6 | Ablation analysis of the top-performing `BBKNN (TS)` method. The `BBKNN (TS)` method performed standard linear expression scaling to 10^4 total counts followed by `log1p` transformation. It then applied three additional transforms: “Standardize” called `sc.pp.scale` to further scale the data to mean 0 and unit variance, “ComBat+PCA” called `sc.pp.combat` followed by `sc.tl.pca` to generate the expression embedding, and “BBKNN” applied an implementation of batch-balanced k-nearest neighbors writted by our method. Bars here show the overall performance in the OpenProblems datasets for ablations that include one or more of these components. For each ablation that includes the “BBKNN” component, comparison of the written BBKNN implementation (“Tree search”) and the `bbknn` package implementation[86] (“Package”) is shown. Black dots show individual performance of three replicates of each method.



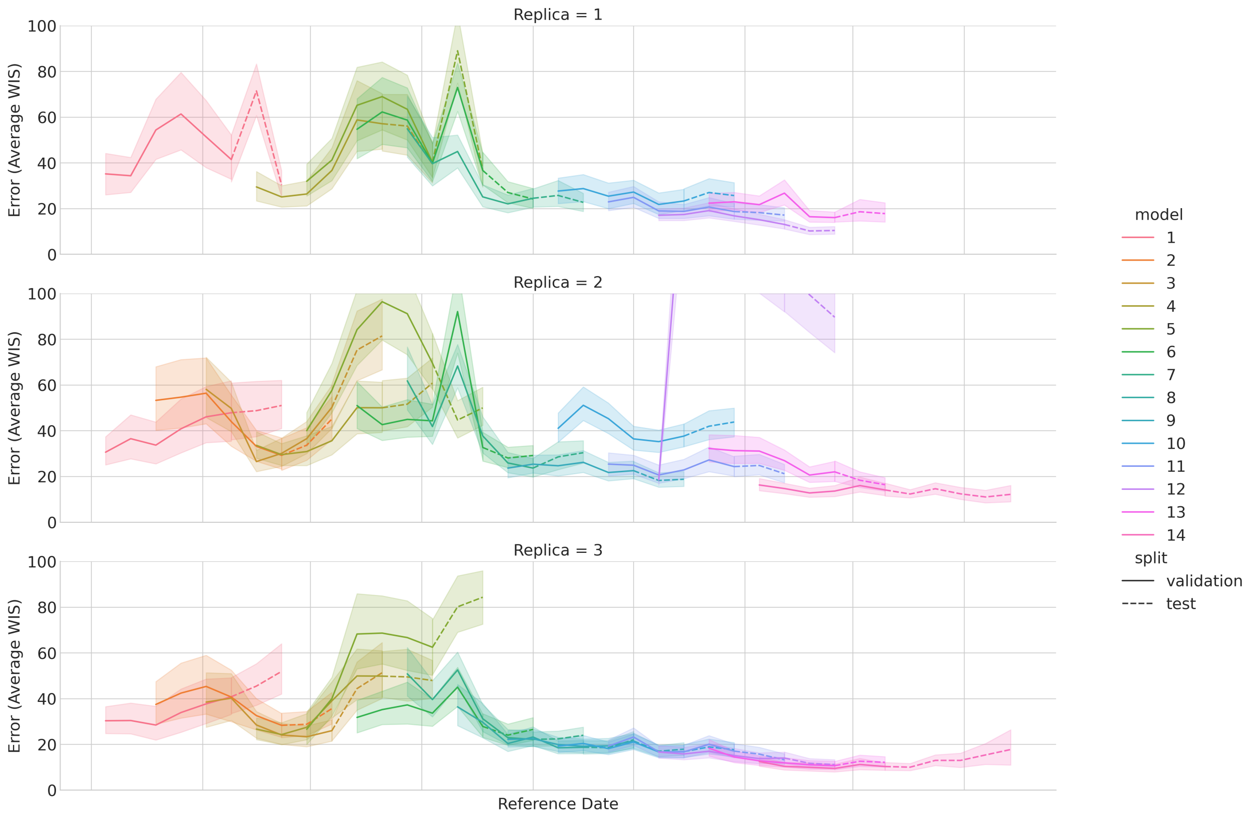
Supplementary Fig. 7 | Comparison of tree search performance on base methods and their "recombination" over an intersection of successfully calculated metrics. We ran "recombination" experiments by seeding tree search with the top variants from two base method runs (see Methods). We compare the performance of two base methods and "recombination" on the OpenProblems test dataset for all 55 pairwise combinations of the 11 base methods. Since sometimes methods may fail getting a score for certain evaluation metrics due to errors like out of memory, we compare the performance on a subset of metrics that were successfully computed for all three methods. "n=X/78" on each subplot shows the number of successfully computed metrics, X, that we averaged over. For each subplot, we show the base methods on the left in light blue, and the recombination method on the right (labeled as "Recomb"), where a green bar means the recombination method outperforms both of its base methods, dark blue means the recombination method outperforms one of the base methods, and red means the recombination method does not outperform either of the base methods.



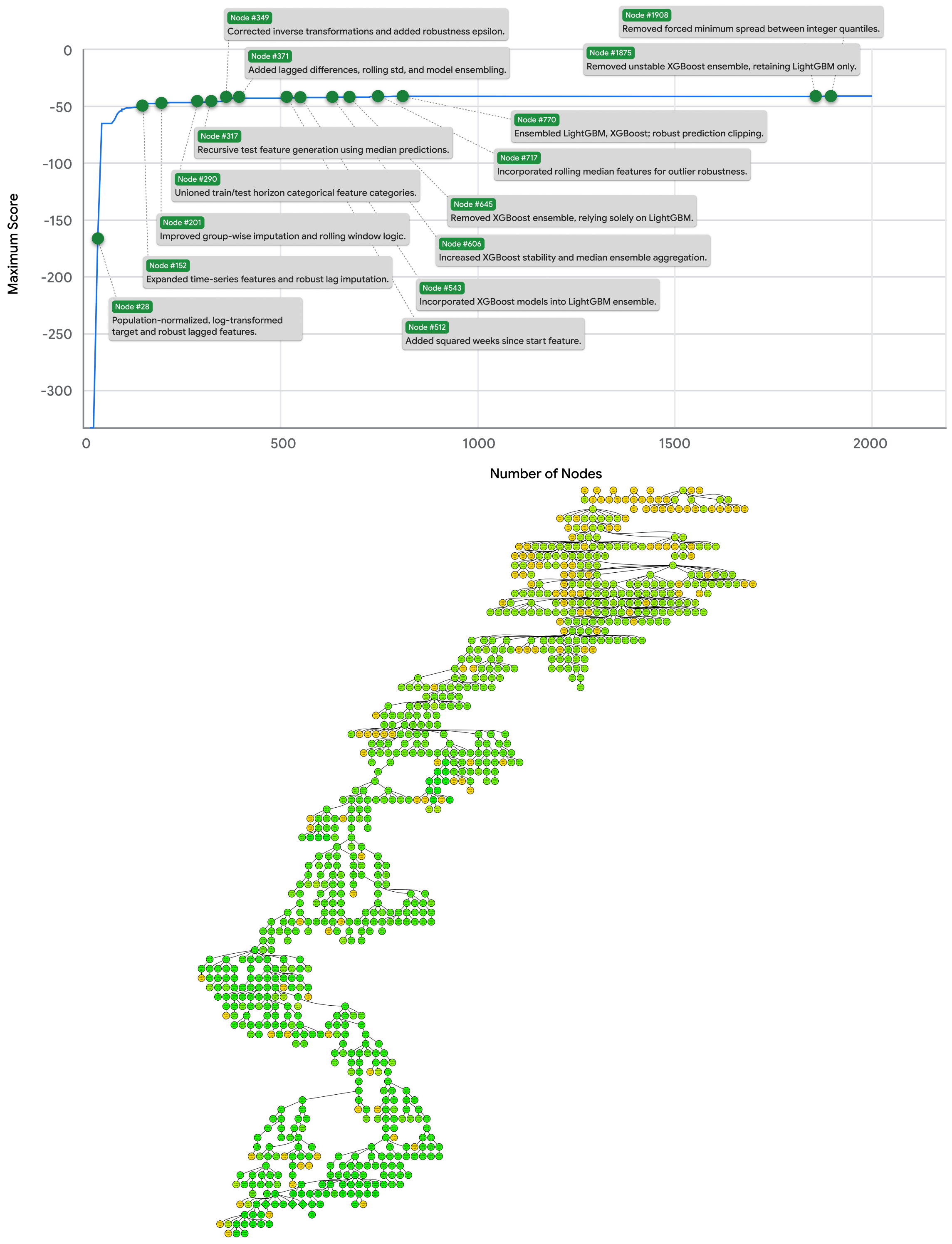
Supplementary Fig. 8 | Heatmap of text embedding cosine similarities among tree search-generated methods. The similarity matrix was hierarchically clustered along rows and columns and reordered to group similar methods together. Three distinct color bars denote major method categories. The pairwise cosine similarities between tree search-generated solutions were greater than 0.85. For context, the lower bound of cosine similarity, established by averaging the similarities between GIFT-Eval's methods (a completely different benchmark) and batch integration methods, was 0.74.



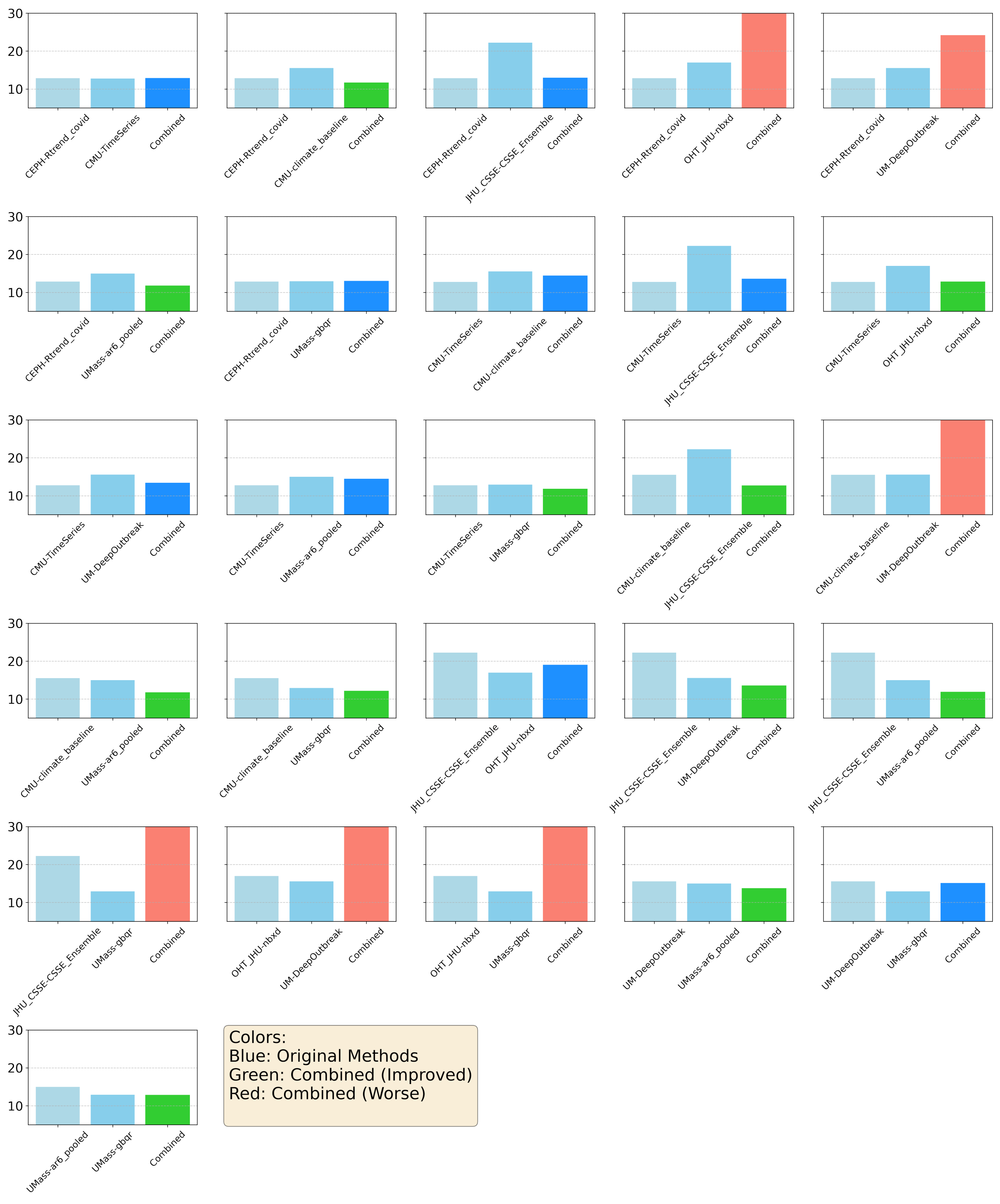
Supplementary Fig. 9 | Performance of the best retrospective COVID-19 hospitalization forecast replicates. This figure presents WIS by reference date for the single best-performing replicate of each validation window in our retrospective COVID-19 forecasting study. The best models are selected based on their performance on the validation dates. The plot shows how finding optimum models on a handfull of validation dates (6 weeks) generalizes on the next two weeks of unseen reference dates.



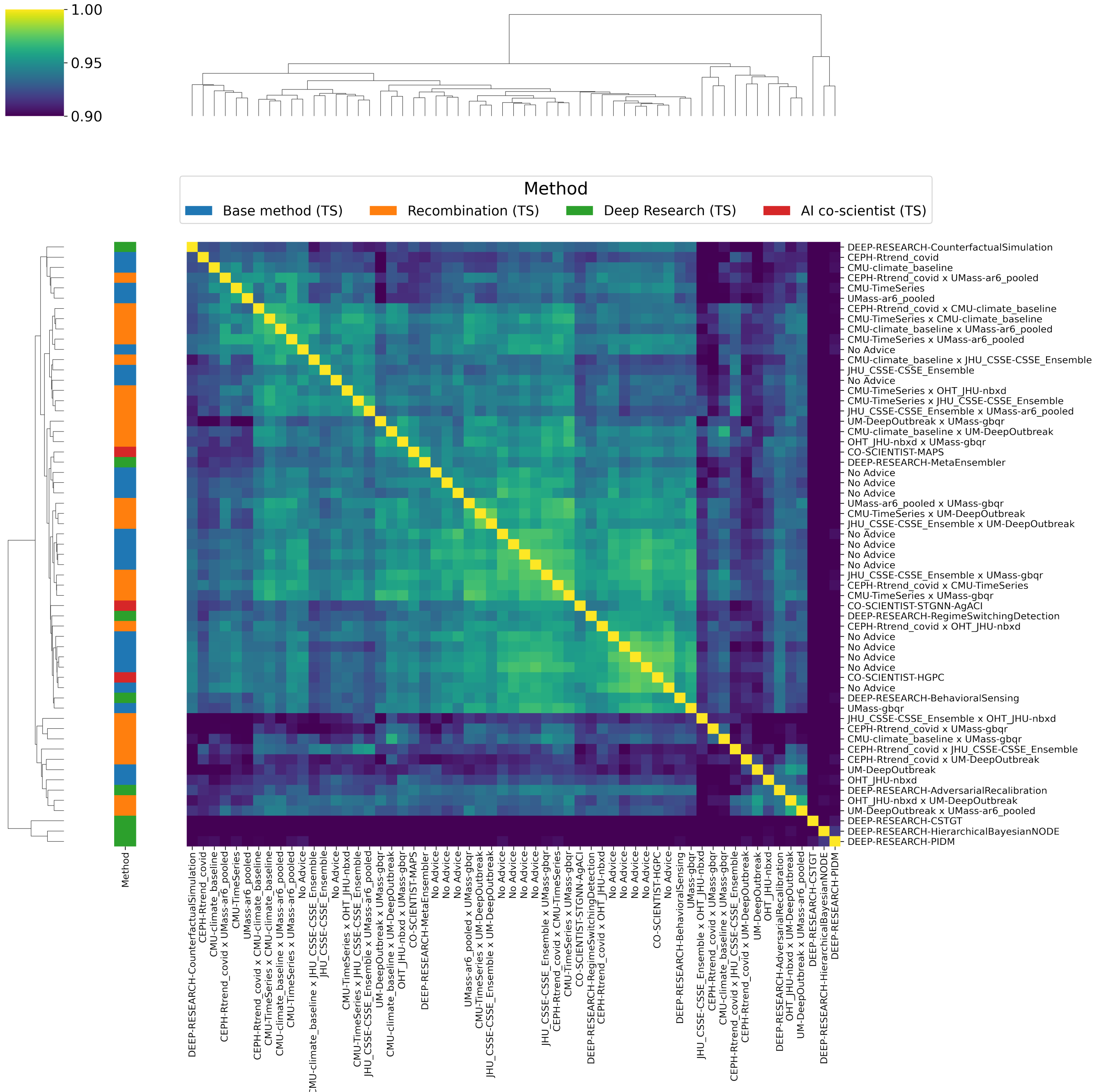
Supplementary Fig. 10 | Performance of retrospective COVID-19 hospitalization forecasts across all replicates. Each panel displays the average WIS by reference date for individual replicates of our proposed models, for all rolling validation dates. Lower WIS values indicate superior forecasting accuracy and calibration. The consistent trends across replicates demonstrate the robustness and reproducibility of tree search's ability to generate high-performing probabilistic forecasts.



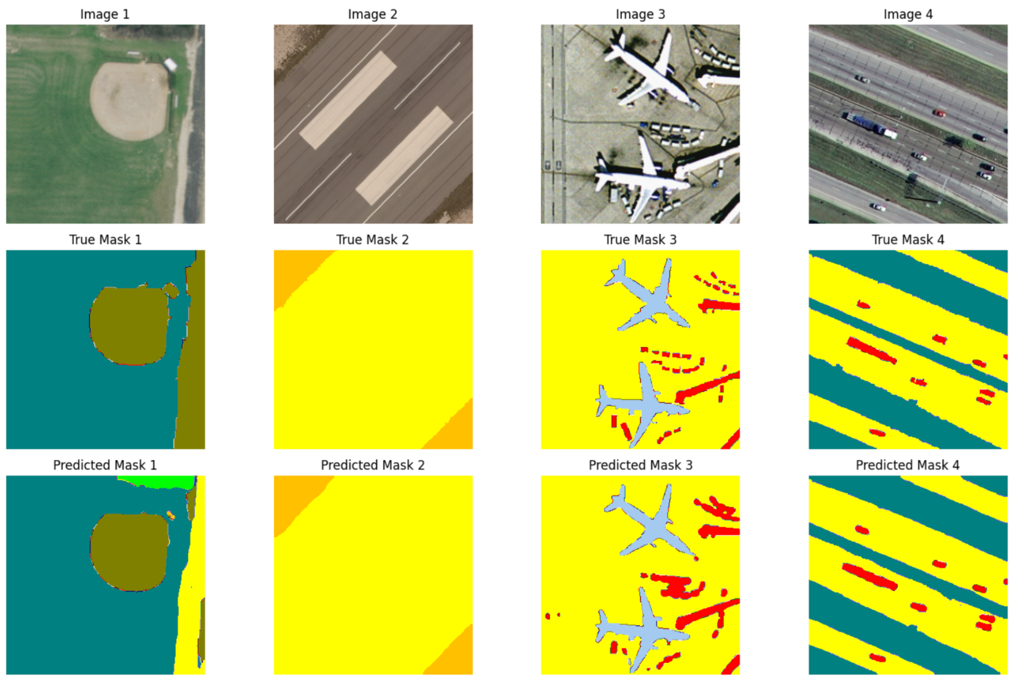
Supplementary Fig. 11 | *Top Figure* Breakthrough plot for the retrospective COVID-19 prediction, showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.



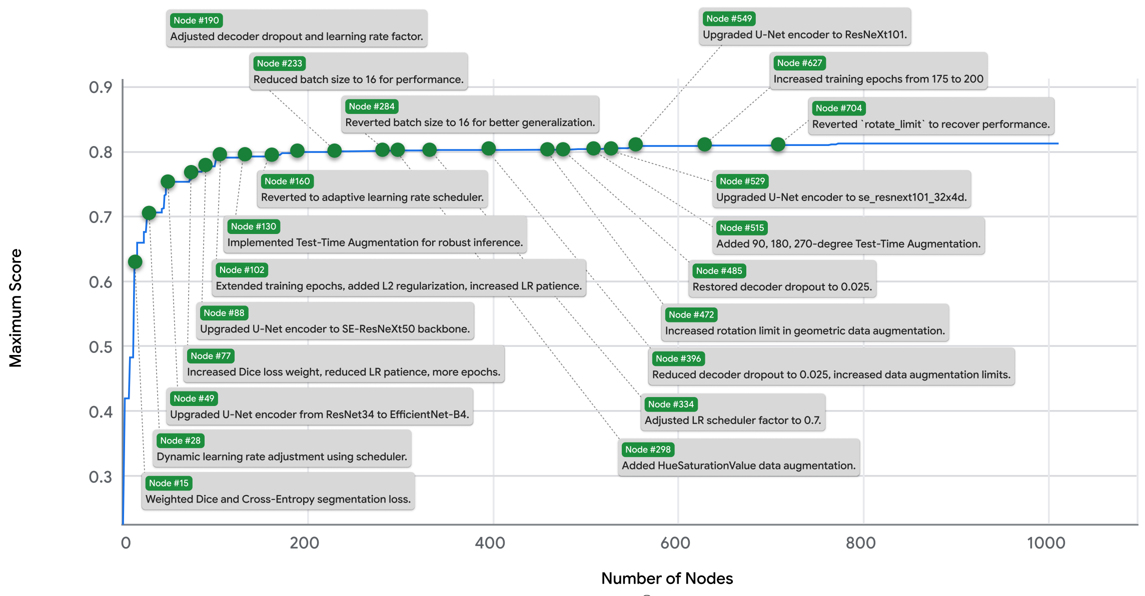
Supplementary Fig. 12 | Performance of recombination experiments for COVID-19 forecasting. This series of bar plots illustrates the average WIS achieved by various hybrid models (right bar, labeled "Recomb") compared to their constituent baseline models (left bars, typically light blue) from the CovidHub competition. Each subplot represents a recombination experiment, demonstrating the success of our system in synthesizing novel forecasting strategies. Green bars indicate that the recombination outperformed both parent models, dark blue indicates it outperformed one, and red indicates it outperformed neither. These results emphasize the search system's ability to combine the strengths of existing methodologies to achieve superior predictive performance.

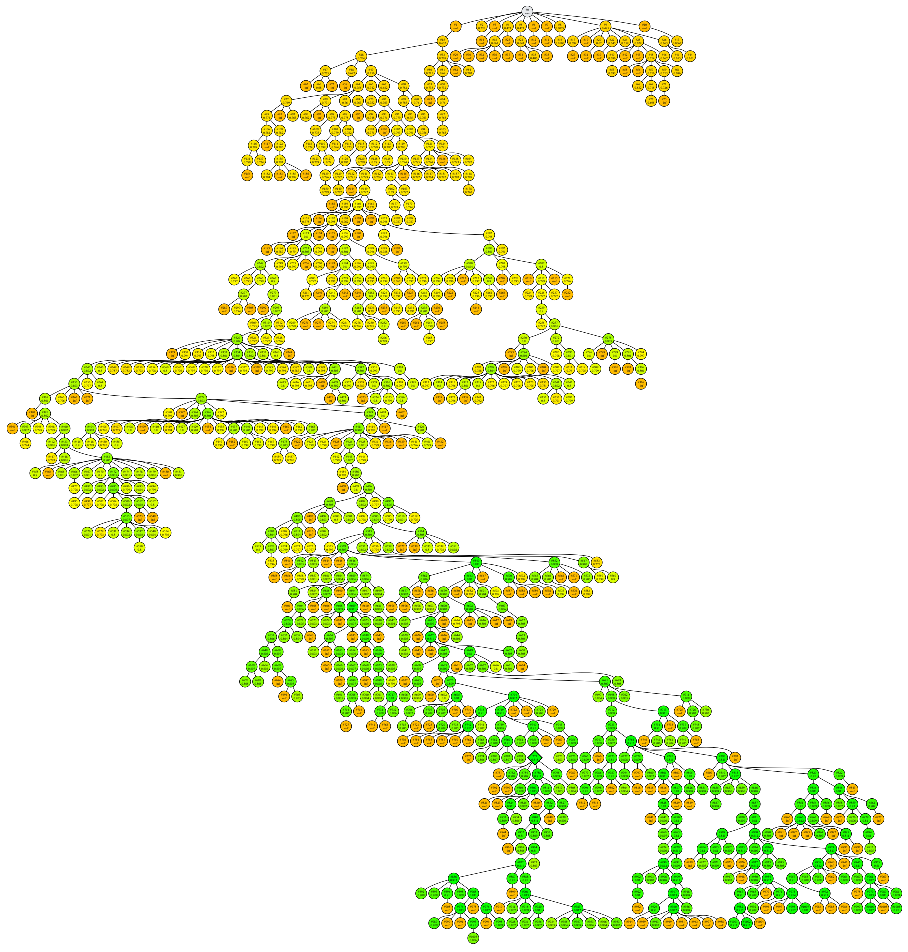


Supplementary Fig. 13 | Heatmap of conceptual similarities among COVID-19 forecasting generated codes for methods. This figure displays the pairwise cosine similarities between text embeddings of all forecasting models generated by tree search for the COVID-19 prediction task. Text embeddings were produced using a Gemini model [79]. The similarity matrix was then hierarchically clustered and reordered to group conceptually related strategies. The color-coded sidebar categorizes each method by its origin illustrating the composition of the emergent conceptual clusters. The No Advice methods are from the Google Retrospective study.

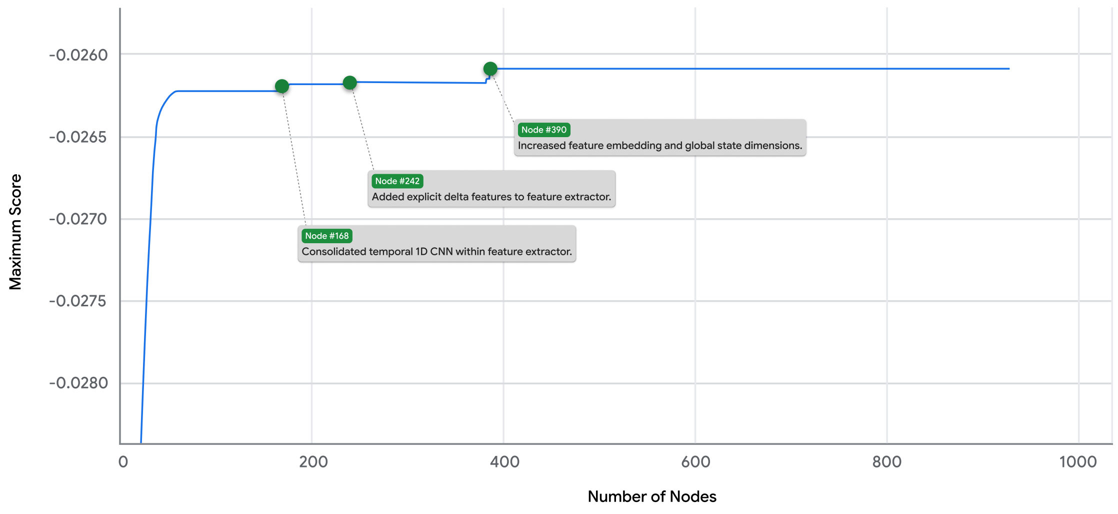


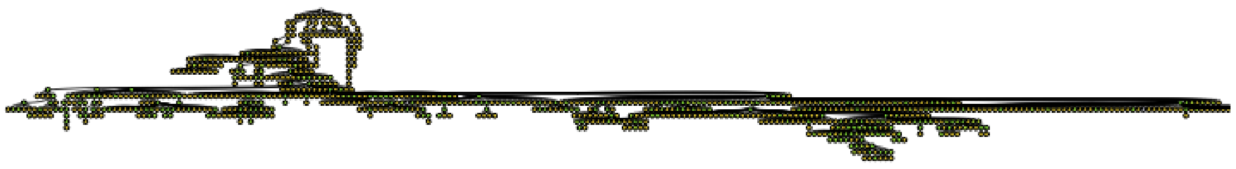
Supplementary Fig. 14 | Example output segmenting DLRSD image pixels from our method Solution 1 (U-Net++).



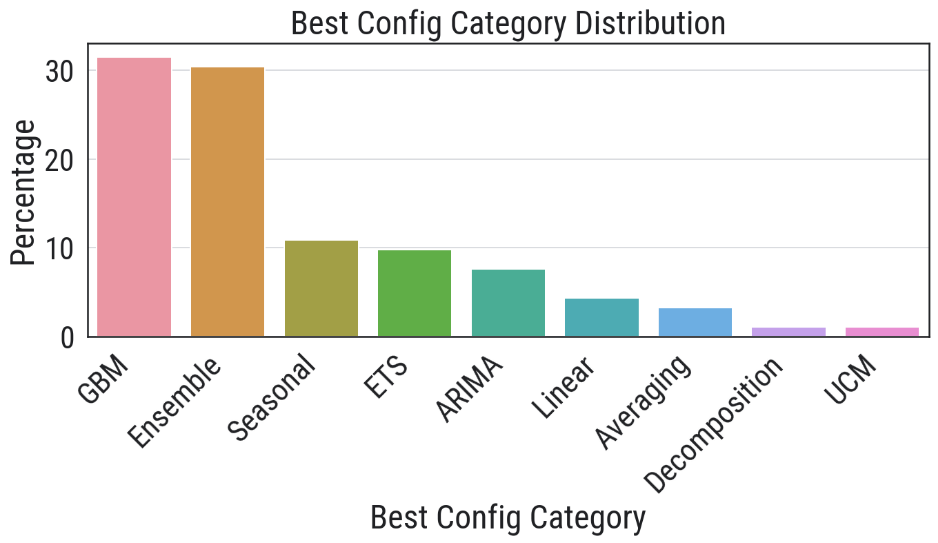


Supplementary Fig. 15 | *Top Figure* Breakthrough plot for the U-Net Geospatial DLRSD solution (solution 3), showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.

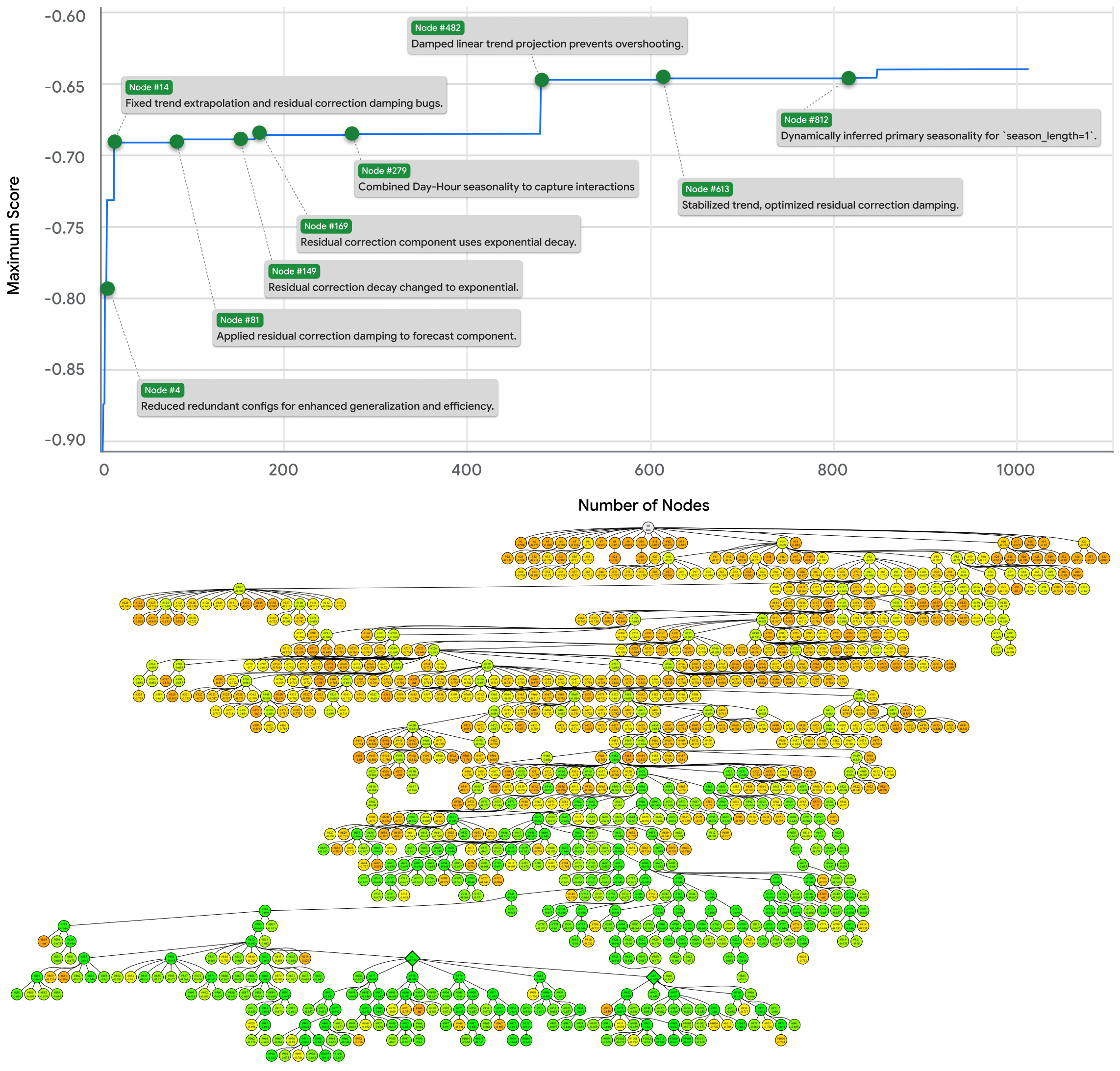




Supplementary Fig. 16 | *Top Figure* Breakthrough plot for the ZAPBench tree search, showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.



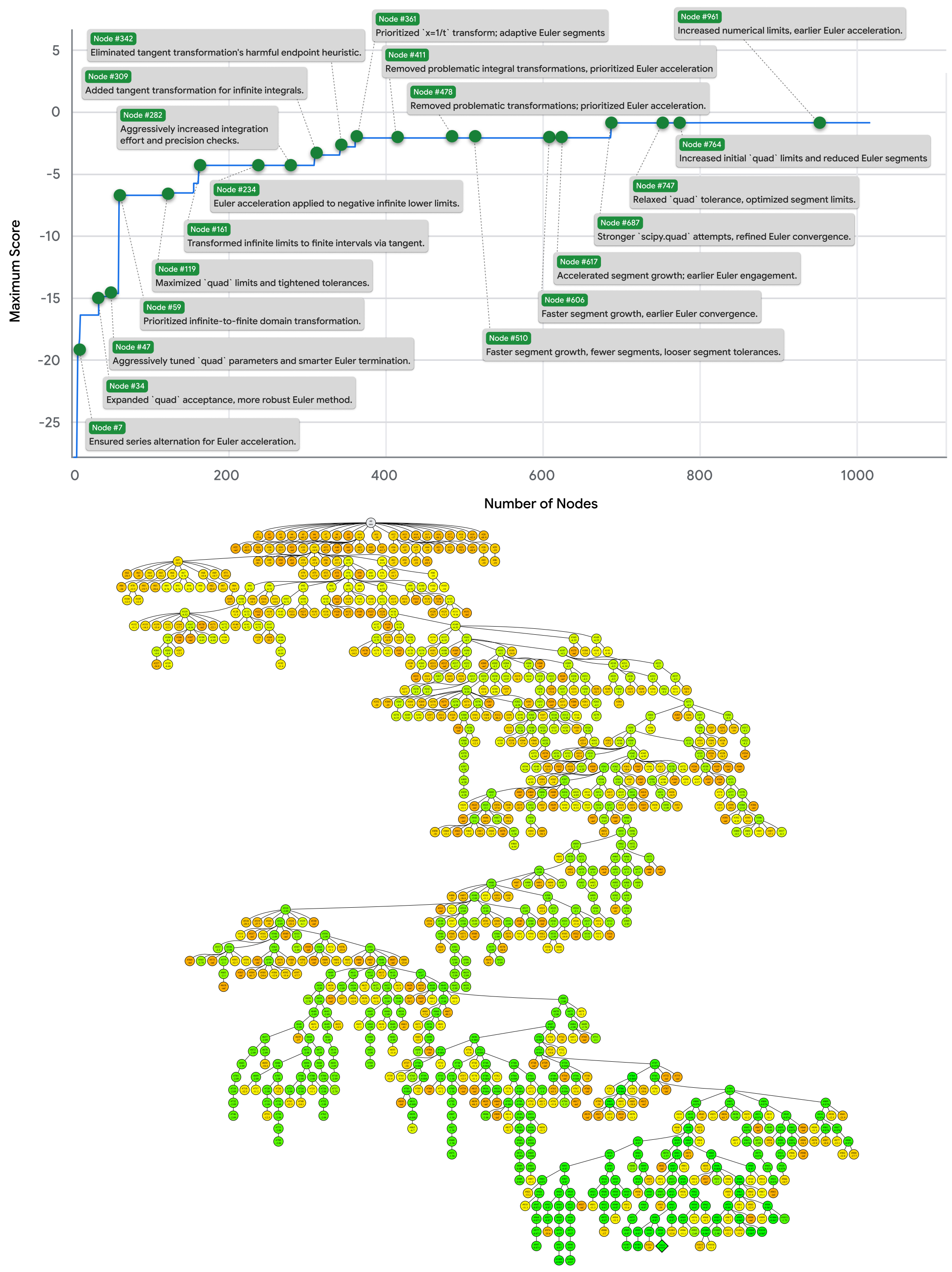
Supplementary Fig. 17 | Categories of solutions on the GIFT-Eval benchmark on the `per-dataset solution (v1)`. We prompted an LLM (Gemini 2.5 Pro) to categorize the code from each of the solutions into a class of methods. The figure shows the percentage of the best codes for each of the 92 competitions in the specified categories: Gradient Boosted Method (GBM); Ensemble; Seasonal; Error, Trend and Seasonality (ETS); Arima[58]; Linear; Averaging; Decomposition and Unobserved components model (UCM).



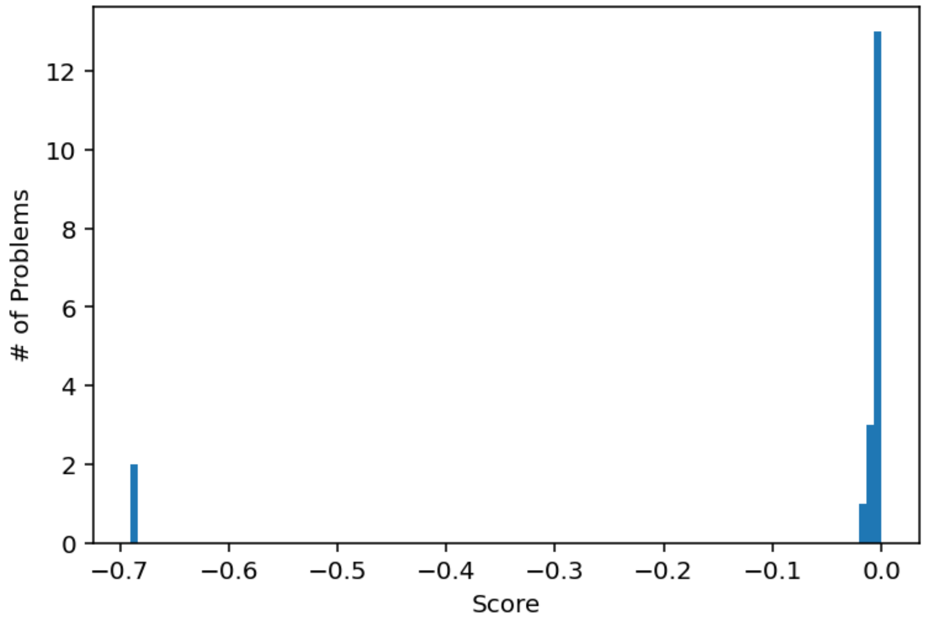
Supplementary Fig. 18 | *Top Figure* Breakthrough plot for the GIFT-Eval tree search, showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.

train set		test set	
445.001	$\int_0^\infty \sin\left(x^2\right) dx$	446.021	$\int_0^\infty \left(\sin^4\left(ax^2\right) - \sin^4\left(bx^2\right)\right) dx$
445.017	$\int_0^\infty \sin\left(ax^2\right) \cos\left(2bx\right) dx$	446.045	$\int_0^\infty x \cos\left(ax^2\right) \cos\left(2bx\right) dx$
447.012	$\int_0^\infty \sin\left(ax^2 + \frac{b^2}{a}\right) \cos\left(2bx\right) dx$	449.013	$\int_0^\infty x^{\mu-1} \sin\left(ax\right) \cos\left(bx\right) dx$
458.031	$\int_0^\infty \left(\frac{\gamma+x}{\beta^2+(\gamma+x)^2} - \frac{\gamma-x}{\beta^2+(\gamma-x)^2}\right) \sin\left(ax\right) dx$	465.002	$\int_0^\infty \frac{\left(3-4\sin^2\left(ax\right)\right)\sin^2\left(ax\right)}{x} dx$
462.034	$\int_0^\infty \frac{x \sin\left(ax\right)\cos\left(bx\right)}{c^2+x^2} dx$	465.013	$\int_0^\infty \frac{\sin^{2m+1}\left(x\right)\sin\left(x(6m+3)\right)}{a^2+x^2} dx$
477.049	$\int_0^\infty \frac{x \sin\left(ax\right)+\cos\left(ax\right)}{x^2+1} dx$	467.025	$\int_0^\infty \frac{\sin\left(x\right)\cos\left(x\right)}{x\sqrt{\sin^2\left(x\right)+1}} dx$
478.036	$\int_0^\infty \frac{\left(\cos\left(a\right)-\cos\left(anx\right)\right)\sin\left(mx\right)}{x} dx$	478.031	$\int_0^\infty \sin\left(ax^p\right) dx$
487.011	$\int_0^\infty \frac{1}{x} \frac{\sin\left(x\right)}{\left(a^2\cos^2\left(x\right)+b^2\sin^2\left(x\right)\right)^2} dx$	478.050	$\int_u^\infty \frac{\cos\left(ax\right)}{\sqrt{-u+x}} dx$
487.026	$\int_0^\infty \frac{1}{x} \frac{\sin\left(x\right)\cos^2\left(x\right)}{\left(a^2\cos^2\left(x\right)+b^2\sin^2\left(x\right)\right)^2} dx$	484.059	$\int_0^\infty \left(\sin\left(a-x^2\right) + \cos\left(a-x^2\right)\right) dx$
488.014	$\int_0^\infty \frac{1}{x} \frac{\sin^3\left(x\right)\cos\left(x\right)}{\left(a^2\cos^2\left(2x\right)+b^2\sin^2\left(2x\right)\right)^4} dx$	487.068	$\int_0^\infty \frac{\cos\left(x\right)\cos\left(a\cos\left(x\right)\right)\cos\left(2nx\right)\sinh\left(a\sin\left(x\right)\right)}{x} dx$
491.004	$\int_0^\infty \frac{\cos^{2m}\left(x\right)}{a^2+x^2} dx$	494.006	$\int_0^\infty x \sin\left(2bx\right) \cos\left(ax^2\right) dx$
491.006	$\int_0^\infty \frac{\cos^{2m+1}\left(x\right)}{a^2+x^2} dx$	496.037	$\int_0^\infty \frac{\sin^3\left(x\right)}{\left(a^2\cos^2\left(x\right)+b^2\sin^2\left(x\right)\right)^3} \frac{1}{x} dx$
491.014	$\int_0^\infty \frac{x \sin\left(2ax\right)\cos^2\left(bx\right)}{\beta^2+x^2} dx$	504.025	$\int_0^\infty \frac{\sin\left(ax^p\right)}{x} dx$
493.056	$\int_0^\infty \frac{\sin\left(2ax\right)\cos^2\left(bx\right)}{x} dx$	504.061	$\int_0^\infty \frac{\sin^3\left(x\right)\cos\left(x\right)}{x\sqrt{\sin^2\left(2x\right)+1}} dx$
495.029	$\int_0^\infty \frac{\sin^3\left(ax\right)\sin^2\left(bx\right)}{x} dx$	505.006	$\int_0^\infty \frac{\sqrt{-b+\sqrt{b^2+x^2}}\sin\left(ax\right)}{\sqrt{b^2+x^2}} dx$
504.057	$\int_0^\infty \frac{\sin^3\left(x\right)\cos\left(x\right)}{x\sqrt{\cos^2\left(2x\right)+1}} dx$	505.008	$\int_0^\infty \frac{\sin\left(x\right)}{x\left(a^2\sin^2\left(x\right)+b^2\cos^2\left(x\right)\right)} dx$
512.029	$\int_0^\infty \frac{\cos\left(bx\right)\cos\left(p\sqrt{a^2+x^2}\right)}{c^2+x^2} dx$	505.023	$\int_0^\infty \frac{\left(\cos\left(a\right)-\cos\left(anx\right)\right)\sin\left(mx\right)}{x} dx$
512.037	$\int_0^\infty \frac{\cos\left(bx\right)\cos\left(p\sqrt{a^2+x^2}\right)}{a^2+x^2} dx$	513.033	$\int_0^\infty \frac{\sin^3\left(ax\right)\cos\left(3bx\right)}{x^2} dx$
550.003	$\int_0^\infty \frac{\sin\left(ax\right)\coth\left(\frac{\pi x}{2}\right)}{x^2+1} dx$	551.027	$\int_0^\infty \frac{\sin^3\left(a^2x^2\right)}{x^2} dx$

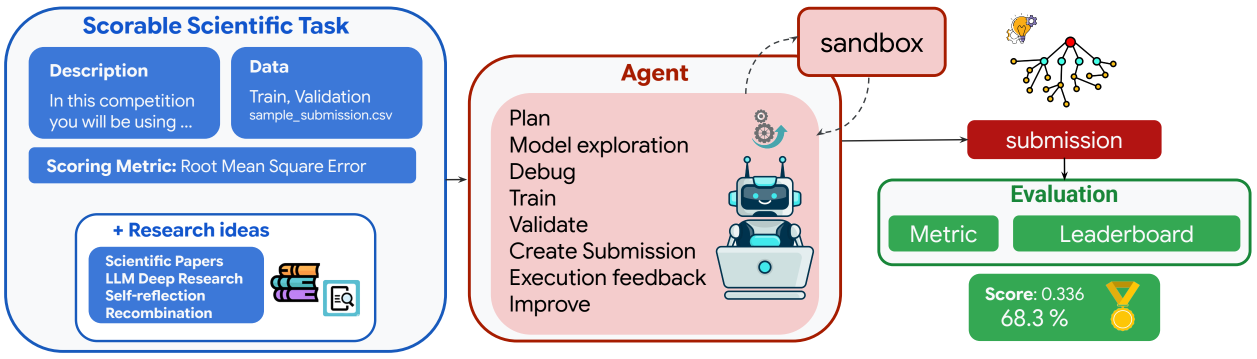
Supplementary Fig. 19 | The dataset of 38 definite integrals with oscillatory integrands on semi-infinite domains[60], none of which were solved correctly by `scipy.integrate.quad()`. Parameters like a, b, c were chosen randomly between 0 and 5 with exponents constrained to be integers.



Supplementary Fig. 20 | *Top Figure* Breakthrough plot for the Integral tree search, showing the evolution of the maximum score as a function of the number of nodes. The green dots label places where the score abruptly increases due to an improvement in the code, and the label describes the change in the code that resulted in the score increase. *Bottom Figure* Structure of the tree for this same search. The color range consists of orange (lower scores) to green (higher scores) with the highest score denoted by a diamond node.



Supplementary Fig. 21 | Scores of the best numerical integration routine applied to the held-out set of 19 integrals. Zero is a perfect score. The generated function solved 17 of 19 integrals to within 3 percent. The standard function, `scipy.integrate.quad()` failed in all these cases.



Supplementary Fig. 22 | *Schematic of Algorithm*, consisting of a code mutation system, where the prompt is augmented with research ideas. Research ideas can be sourced from the primary literature, or from a search algorithm.

Supplementary Tables

Supplementary Table 1 | Basic Prompt Playground Competitions. The prompt is used for the TS on the Kaggle Playground Benchmark. This example is for Season 3 Episode 17.

Prompt for Kaggle Playground Competitions

Please write the python code to work on a Kaggle competition. Use any model you like.
Kaggle competition name: Binary Classification of Machine Failures
The competition is evaluated as follows: Submissions are evaluated on area under the ROC curve between the predicted probability and the observed target.

~~~
Submission File
For each `id` in the test set, you must predict the probability of a `Machine failure`.
The file should contain a header and have the following format:

    id,Machine failure
    136429,0.5
    136430,0.1
    136431,0.9
    etc.

Here are a few lines of each of the files:

file_name : sample_submission.csv

file_contents:

id,Machine failure
79996,0
100009,0
etc.

====
file_name : test.csv

file_contents:
etc.
====
file_name : train.csv

file_contents:
etc.
====
Please provide complete code that will generate the submission file in the
format below:

```python
YOUR CODE
```
~~~

Supplementary Table 2 | Expert Advice for Playground Competitions. The prompt is used for the TS with Expert Advice on the Kaggle Playground Benchmark.

Expert Advice Prompt for Kaggle Playground Competitions

Here is high level advice: Instead of putting all your effort into a single model, experiment with combining two or more models. Start with simple averaging of predictions and then explore more advanced techniques like stacking.
Try out several different types of models (e.g., gradient boosting machines, linear models, and even simpler models like logistic regression) to see how they perform.
Look for opportunities to go beyond standard preprocessing. Investigate the data for potential leaks, and consider using optimization libraries to find the best way to combine your models' predictions.
While feature engineering is a crucial skill, it's also important to recognize when it might not be the most important factor. Sometimes, the choice of model and ensembling strategy can have a bigger impact. Don't be afraid to try a more "brute-force" approach with powerful models that can handle raw data effectively.

Supplementary Table 3 | Boosted Decision trees for Playground Competitions. The prompt is used for the TS with Boosted Decision Tree on the Kaggle Playground Benchmark.

Expert Advice Prompt for Kaggle Playground Competitions

Given the code you are given please rewrite any library code (such as XGBoost, LightGBM, and CatBoost) by making internal algorithmic choices that produce performant training code and models that generalize well in many situations. Things you can try are alternative representations of data, using different step size algorithms, using the output of a strong learner as input to the next weak learner. If the code contains such libraries, please extract the raw code that is being used in the library and rewrite it to improve performance.

Supplementary Table 4 | Example code generated by tree search. We prompted tree search to generate a solution for single-cell batch integration. We refer to this solution as `BBKNN (TS)`.

Example code generated by our method.

```python
# [rewrite_cell]
from typing import Any
from sklearn.decomposition import TruncatedSVD
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import lil_matrix, csr_matrix
import numpy as np
import scanpy as sc
import anndata as ad
import heapq # For efficiently getting top K elements from merged lists

# Define parameters for the config.
# These values are chosen to balance computational cost and integration performance
# for datasets with up to ~300k cells and 2k genes.
config = {
    'n_pca_components': 100,     # Number of PCA components. Recommended: 50-200.
                                 # Captures sufficient variance while reducing dimensionality.
    'n_neighbors_per_batch': 10, # Number of neighbors to find within each batch. Recommended: 5-15.
                                 # This defines the local batch context for each cell.
    'total_k_neighbors': 50,     # Total number of nearest neighbors to retain for the final graph. Recommended: 15-100.
                                 # This forms the global batch-integrated graph.
}


def eliminate_batch_effect_fn(
    adata: ad.AnnData, config: dict[str, Any]
) -> ad.AnnData:
  # Create a copy to ensure the original input adata remains unchanged.
  adata_integrated = adata.copy()

  # --- Preprocessing: Normalize, log-transform, scale ---
  # These are standard initial steps for scRNA-seq data.
  # Use adata.X which contains raw counts.
  sc.pp.normalize_total(adata_integrated, target_sum=1e4)
  sc.pp.log1p(adata_integrated)
  sc.pp.scale(adata_integrated, max_value=10) # Clip values to avoid extreme outliers

  # --- Batch Correction: ComBat on the gene expression matrix ---
  # This step applies a more robust linear model-based batch correction
  # directly on the gene expression data before dimensionality reduction.
  # ComBat modifies adata_integrated.X in place.
  sc.pp.combat(adata_integrated, key='batch')

  # --- Dimensionality Reduction: PCA on the ComBat-corrected data ---
  # n_comps cannot exceed min(n_obs - 1, n_vars). Robustly handle small datasets.
  n_pca_components = config.get('n_pca_components', 100)
  actual_n_pca_components = min(n_pca_components, adata_integrated.n_vars, adata_integrated.n_obs - 1)

  # Handle edge cases for PCA and graph construction where data is too small.
  # If PCA cannot be run meaningfully, return a minimal AnnData object to avoid errors.
  if actual_n_pca_components <= 0 or adata_integrated.n_obs <= 1:
      print(f"Warning: Too few observations ({adata_integrated.n_obs}) or dimensions ({adata_integrated.n_vars}) for PCA/graph construction. Returning trivial embedding.")
      # Provide a placeholder embedding and empty graph structure.
      adata_integrated.obsm['X_emb'] = np.zeros((adata_integrated.n_obs, 1))
      adata_integrated.obsp['connectivities'] = csr_matrix((adata_integrated.n_obs, adata_integrated.n_obs))
      adata_integrated.obsp['distances'] = csr_matrix((adata_integrated.n_obs, adata_integrated.n_obs))
      adata_integrated.uns['neighbors'] = {
          'params': {
              'n_neighbors': 0,
              'method': 'degenerate',
              'n_pcs': 0,
              'n_neighbors_per_batch': 0,
              'pca_batch_correction': 'none',
          },
          'connectivities_key': 'connectivities',
          'distances_key': 'distances',
      }
      return adata_integrated

  sc.tl.pca(adata_integrated, n_comps=actual_n_pca_components, svd_solver='arpack')

  # Set the ComBat-corrected PCA embedding as the integrated output embedding.
  # This 'X_emb' will be directly evaluated by metrics like ASW, LISI, PCR.
  adata_integrated.obsm['X_emb'] = adata_integrated.obsm['X_pca']


  # --- Custom Batch-Aware Nearest Neighbors Graph Construction ---
  # This implements the expert advice: find neighbors independently within batches, then merge.
  # This part of the code remains largely the same, but now operates on the
  # ComBat-corrected PCA embedding (adata_integrated.obsm['X_emb']).
  k_batch_neighbors = config.get('n_neighbors_per_batch', 10)
  total_k_neighbors = config.get('total_k_neighbors', 50)

  # A list of dictionaries to store unique neighbors and their minimum distances for each cell.
  # Using dictionaries allows efficient updating if a cell is found as a neighbor from multiple batches.
  merged_neighbors_per_cell = [{} for _ in range(adata_integrated.n_obs)]

  # Group cell indices by batch for efficient querying.
  batches = adata_integrated.obs['batch'].values
  unique_batches = np.unique(batches)
  batch_to_indices = {b: np.where(batches == b)[0] for b in unique_batches}

  # Pre-fit NearestNeighbors models for each batch's data using the corrected PCA embedding.
  # This avoids refitting the model for every query.
  batch_nn_models = {}
  for b_id in unique_batches:
    batch_cell_indices = batch_to_indices[b_id]
    # Ensure there are enough cells to fit a NearestNeighbors model (at least k_batch_neighbors + 1 for self-exclusion, or just > 0 for min k=1)
    if len(batch_cell_indices) > 0:
        # Fit with a k that is at most the batch size to avoid errors if k_batch_neighbors is too high for a small batch.
        k_fit_effective = min(k_batch_neighbors + 1, len(batch_cell_indices)) # +1 to ensure self-loop can be found and excluded
        if k_fit_effective > 0: # Only fit if there are points available
            nn_model = NearestNeighbors(n_neighbors=k_fit_effective, metric='euclidean', algorithm='auto')
            nn_model.fit(adata_integrated.obsm['X_emb'][batch_cell_indices])
            batch_nn_models[b_id] = nn_model
```

Example code generated by our method (continued).

```python

# Iterate through all possible query batches and target batches to find neighbors.
for query_batch_id in unique_batches:
    query_global_indices = batch_to_indices[query_batch_id]
    if len(query_global_indices) == 0:
        continue # Skip empty query batches

    query_data = adata_integrated.obsm['X_emb'][query_global_indices]

    for target_batch_id in unique_batches:
        if target_batch_id not in batch_nn_models:
            continue # Skip target batches that were too small to fit an NN model

        nn_model = batch_nn_models[target_batch_id]
        target_global_indices = batch_to_indices[target_batch_id]

        # Ensure n_neighbors does not exceed the number of points in the target batch.
        k_for_query = min(k_batch_neighbors, len(target_global_indices) -1) # -1 to avoid finding self as neighbor if batch is query batch
        if k_for_query <= 0: # No valid neighbors can be found in this target batch
          continue

        # Query neighbors for all cells in the current query batch against the target batch's data.
        distances, indices_in_target_batch = nn_model.kneighbors(query_data, n_neighbors=k_for_query, return_distance=True)

        for i_query_local in range(len(query_global_indices)):
            current_cell_global_idx = query_global_indices[i_query_local]

            dists_for_cell = distances[i_query_local]
            global_neighbors_for_cell = target_global_indices[indices_in_target_batch[i_query_local]]

            for k_idx in range(len(global_neighbors_for_cell)):
                neighbor_global_idx = global_neighbors_for_cell[k_idx]
                dist = dists_for_cell[k_idx]

                # Exclude self-loops: a cell should not be its own neighbor in graph construction.
                if neighbor_global_idx == current_cell_global_idx:
                    continue

                # Store neighbor and its distance. If already present, keep the minimum distance (closest connection).
                if (neighbor_global_idx not in merged_neighbors_per_cell[current_cell_global_idx] or
                    dist < merged_neighbors_per_cell[current_cell_global_idx][neighbor_global_idx]):
                    merged_neighbors_per_cell[current_cell_global_idx][neighbor_global_idx] = dist

# Convert collected neighbors and distances into sparse matrices.
rows = []
cols = []
data_distances = []

for i in range(adata_integrated.n_obs):
  # Retrieve all candidate neighbors for cell 'i', sort by distance, and take the top 'total_k_neighbors'.
  current_cell_candidates = list(merged_neighbors_per_cell[i].items())

  if not current_cell_candidates: # If a cell has no valid neighbors after all filtering
      continue

  # Use heapq for efficient selection of the smallest distances.
  selected_neighbors = heapq.nsmallest(total_k_neighbors, current_cell_candidates, key=lambda item: item[1])

  for neighbor_idx, dist in selected_neighbors:
      rows.append(i)
      cols.append(neighbor_idx)
      data_distances.append(dist)

# Create distance matrix. Handle case with no neighbors found at all for the entire dataset.
if not rows:
    distances_matrix = csr_matrix((adata_integrated.n_obs, adata_integrated.n_obs))
else:
    distances_matrix = csr_matrix((data_distances, (rows, cols)), shape=(adata_integrated.n_obs, adata_integrated.n_obs))

# Symmetrize the distance matrix: if A is a neighbor of B, then B is also a neighbor of A,
# with the distance being the maximum of the two observed distances (ensures undirected graph).
distances_matrix = distances_matrix.maximum(distances_matrix.T)
distances_matrix.eliminate_zeros() # Remove any explicit zeros created by max operation

# Create connectivities matrix (binary representation of connections).
connectivities_matrix = distances_matrix.copy()
connectivities_matrix.data[:] = 1.0  # All non-zero entries become 1.0 (connected).
connectivities_matrix.eliminate_zeros()
connectivities_matrix = connectivities_matrix.astype(float)

# Store the custom graph in adata.obsp. These keys are used by scib metrics.
adata_integrated.obsp['connectivities'] = connectivities_matrix
adata_integrated.obsp['distances'] = distances_matrix

# Store parameters in adata.uns['neighbors'] for completeness and scanpy/scib compatibility.
adata_integrated.uns['neighbors'] = {
    'params': {
        'n_neighbors': total_k_neighbors,
        'method': 'custom_batch_aware_combat_pca', # Reflects the integration strategy
        'metric': 'euclidean',
        'n_pcs': actual_n_pca_components,
        'n_neighbors_per_batch': k_batch_neighbors,
        'pca_batch_correction': 'combat', # Indicates ComBat was applied before PCA
    },
    'connectivities_key': 'connectivities',
    'distances_key': 'distances',
}

return adata_integrated
```

Supplementary Table 5 | Expert manual inspection of adherence of tree search implementation to method.

Method	Replicate	Judgment	Notes
batchelor fastMNN	0	Follow	
batchelor fastMNN	1	Follow	
batchelor fastMNN	2	Follow	
batchelor mnnCorrect	0	Follow	
batchelor mnnCorrect	1	Follow	
batchelor mnnCorrect	2	Follow	
BBKNN	0	Follow	Adds distances between batches, performs spectral clustering on the graph. Does not compute connectivities.
BBKNN	1	Follow + Innovative	Standardize + ComBat + PCA for embedding. BBKNN implemented on that embedding.
BBKNN	2	Follow	Corrects the data, computes neighbors, final embedding is UMAP supposedly based on neighbors.
ComBat	0	Follow	
ComBat	1	Follow	
ComBat	2	Follow	
Harmony	0	Follow	Entropy-based diversity penalty.
Harmony	1	Follow	Linear diversity penalty.
Harmony	2	Follow	Linear diversity penalty.
LIGER	0	Follow	Uses `sklearn.NMF` with multiplicative update solver.
LIGER	1	Follow	Writes NMF function from scratch. Builds single global KNN graph rather than by batch.
LIGER	2	Not relevant	Uses ComBat + SVD.
No advice	0	Follow	Uses batch-specific mean+std for all genes to rescale. Then PCA.
No advice	1	Follow	ComBat + SVD
No advice	2	Follow	ComBat + PCA
SCALEX	0	Follow	Adds log_var clipping and weight normalization.
SCALEX	1	Follow	Learns batch embedding. Learns gamma and beta conditioned on batch index. Batch index not supplied to first layer of decoder.
SCALEX	2	Follow	Uses min_delta for robust early stopping. batch_index not supplied to the first layer of the decoder.
Scanorama	0	Follow	
Scanorama	1	Not relevant	Implements mnnpy via `sc.external.pp.mnn_correct`.
Scanorama	2	Follow	
scVI	0	Follow	Applies log1p scaling with ZINB loss. Fits global dispersion theta rather than batch-specific.
scVI	1	Follow	Applies optional log1p scaling with ZINB loss. Fits global dispersion theta rather than batch-specific.
scVI	2	Follow	Expression frequency exponentiated rather than softmaxed. Applies log1p scaling with ZINB loss. Fits global dispersion theta rather than batch-specific.
TabVI	0	Follow	
TabVI	1	Follow	
TabVI	2	Follow	

Supplementary Table 6 | Prompt for recombination of baseline method ideas. The prompt instructs Gemini to identify the main differences in the principles of top-performing solutions, obtained from tree search runs seeded with baseline methods. This generated summary then serves as part of an explicit instruction for tree search to create hybrid strategies.

Prompt for summarizing differences between two baseline methods.

```
Compare these two code solutions to the same problem of integrating single-cell batch effects.
Explain the main principles that differ between the codes:

CODE 1: [CODE FROM BASELINE 1]

CODE 2: [CODE FROM BASELINE 2]
```

Supplementary Table 7 | Method descriptions used for replicating COVID-19 models submitted to the CDC's CovidHub.

`CEPH-Rtrend_covid`

"Use a renewal equation method based on Bayesian estimation of Rt from hospitalization data. Model forecasts should be obtained by using a renewal equation based on the estimated net reproduction number Rt. Apply a lowpass filter to the time series of weekly hospitalizations, then interpolate it to daily resolution. Then use MCMC Metropolis-Hastings sampling to estimate the posterior distribution of Rt based on the filtered data, considering an informed prior on Rt based on COVID-19 literature. The estimated Rt in the last weeks of available data is used to forecast Rt in the upcoming weeks, with a drift term proportional to the current incidence. Finally, use the renewal equation with the posterior distribution and trend of the estimated Rt in the most recent weeks of hospitalization data."

`CMU-TimeSeries`

"Use an ensemble of AR-based time-series models, involving a basic quantile autoregression fit using lagged values of covid-related hospitalization counts (normalized by population). The data should be smoothed in time. Fit the model jointly across all jurisdictions using the most recently available 21 days of training data. Learn each of the 23 quantiles using a separate quantile regression with nonnegativity and quantile sorting constraints applied post hoc."

`CMU-climate_baseline`

"Use an ensemble of historically formed quantiles. Using data from 2022 onwards, this climatological model should use samples from the 7 weeks centered around the target week and reference week to form the quantiles for the target week, as one might use climate information to form a meteorological forecast. To get more variation at some potential issue of generalization, one can form quantiles after aggregating across geographic values as well as years (after converting to a rate based case count). This model should use a simple average of the geo-specific quantiles and the geo-aggregated quantiles."

`JHU_CSSE-CSSE_Ensemble`

"Use a Multi-Pathogen Optimized Geo-Hierarchical Ensemble Framework (MPOG-Ensemble). Forecast state-level COVID-19 hospitalizations using a combination of time series forecasting methods, organized across three hierarchical levels. At the individual state level, forecasts are generated using Holt-Winters Exponential Smoothing. For regional predictions, which group states based on past 2 years covid-19 activity trends identified through the Louvain method, Long Short-Term Memory (LSTM) models are employed. Additionally, a LSTM model that covers all states is implemented. These three-tiered model outputs are integrated, selecting weights based on their recent performance in terms of Mean Absolute Error (MAE) to produce the final prediction."

`OHT_JHU-nbxd`

“Use a neural network that encodes the data inputs using a TCN (Bai et al. 2018) and decodes the result into a forecast using N-BEATS (Oreshkin et al. 2000). This is a residual block type architecture that generates point forecasts from univariate time series data. The network accepts a fixed lookback window of time points as input, and has a set number of output nodes corresponding to the length of the forecast horizon. Extend the network with additional residual blocks that output error variance forecasts (evaluated using a likelihood loss function) which allows generating quantile forecasts, assuming a parametric (gamma) error distribution. Additional predictor variables are incorporated using a temporal convolutional network (TCN; Bai et al. 2018). The TCN accepts one input channel for each predictor time series (or static variable), including past values of the target variable, and outputs a single channel with the same length as the lookback window. The TCN output channel is used as the input to the extended N-BEATS network. Each value in the TCN output sequence is a non-linear combination of the predictor variables at that point and all previous points in the lookback window, which preserves the temporal structure of the input. Forecast is the median of an ensemble of such models with varying lookback window sizes and random initializations."

`UM-DeepOutbreak`

“Use a deep neural network model with conformal predictions. The neural network architecture is a sequence-to-sequence model based on recurrent units and self-attention modules. It is trained in a multi-task setting where each region is considered a task. The uncertainty quantification is conducted post hoc with conformal predictions that follows adaptive conformal inference to adapt to distribution shifts. Spatial correlation is not considered."

`UMass-ar6_pooled`

“Use an autoregressive model with shared coefficients across locations: AR(6) model after fourth root data transform. AR coefficients are shared across all locations. A separate variance parameter is estimated for each location."

`UMass-gbqr`

“Use gradient boosting quantile regression. Do gradient boosting using features summarizing signal activity, properties of the location, information about the timing of forecast creation, and the forecast horizon."

Supplementary Table 8 | Prompt for replicating COVID-19 models submitted to CovidHub by injecting method descriptions as `{method}` into existing tree search prompt.

Prompt for replicating models submitted to CovidHub.

Please write the python code to work on a competition.
`{method}`
I've already loaded the train / test files and split out the x and y parts.
Please provide a new definition for the function below, complete with imports, that will generalize well. However, do not do any cross-validation in here. Your function should expect options to be passed in via the config argument. I'll use cross-validation myself to select which of the options in the `config_list` generalizes best.
`{method}`

```
from typing import Any  # Don't forget this!
import pandas as pd

def fit_and_predict_fn(
   train_x: pd.DataFrame,
   train_y: pd.Series,
   test_x: pd.DataFrame,
   config: dict[str, Any]) -> pd.Series:
    """Make predictions for test_x by modeling train_x to train_y.
    Do not do any cross-validation in here.
    """
    mean_y = np.mean(train_y)
    return pd.Series([mean_y] * len(test_x), index=test_x.index)

    # These will get scored by code that I supply. You'll get back a summary
    # of the performance of each of them.

config_list = [{}]
```

And format it like this:

```
# YOUR CODE
# YOUR config_list
```

Supplementary Table 9 | Expert manual inspection of adherence of tree search implementation to COVID-19 modeling methods.

Method	Judgment	Notes
CEPH-Rtrend_covid x CMU-TimeSeries	Follow	
CEPH-Rtrend_covid x CMU-climate_ baseline	Follow	
CEPH-Rtrend_covid x JHU_CSSE-CSSE_Ensemble	Follow	
CEPH-Rtrend_covid x OHT_JHU-nbxd	Follow	Translates R_t into engineered features (lagged differences, ratios).
CEPH-Rtrend_covid and UM-DeepOutbreak	Follow	Feeds mechanistic-inspired features into GRU-based encoder, predicts quantiles via pinball loss.
CEPH-Rtrend_covid x UMass-ar6_pooled	Follow + Innovate	Simulates from normal distribution in transformed space then inverse transforms to derive quantiles.
CEPH-Rtrend_covid x UMass-gbqr	Follow	Implements mechanistic model components as input features to ML model.
CMU-TimeSeries x CMU-climate_baseline	Follow	AR model with climatological features as predictors.
CMU-TimeSeries x JHU_CSSE-CSSE_Ensemble	Follow	Hierarchical ensemble of QuantReg AR models with performance-based weighting.
CMU-TimeSeries x OHT_JHU-nbxd	Follow	Ensemble of bagged QuantReg AR models.
CMU-TimeSeries x UM-DeepOutbreak	Follow	LightGBM quantile regression models with iterative forecasting + conformal-like calibration.
CMU-TimeSeries x UMass-ar6_pooled	Follow	Ensemble of AR QuantReg models on fourth-root transformed data.
CMU-TimeSeries x UMass-gbqr	Follow	LightGBM quantile models on population-normalized data with (un)smoothed lags + direct multi-horizon prediction.
CMU-climate_baseline x JHU_CSSE-CSSE_Ensemble	Follow	Hierarchical ensemble of climatological models.
CMU-climate_baseline x OHT_JHU-nbxd	Follow	Feeds climatological quantiles into LightGBM to learn directly from seasonal baseline.
CMU-climate_baseline x UM-DeepOutbreak	Follow	LightGBM to predict central trend + climatological model for empirical quantile spreads.
CMU-climate_baseline x UMass-ar6_pooled	Follow	Seasonally-aware method for estimating uncertainty based on empirical quantiles of AR residuals.

Continued on next page

Supplementary Table 9 – continued from previous page

Method	Judgment	Notes
CMU-climate_baseline x UMass-gbqr	Follow	Feeds climatological statistics as features into LightGBM.
JHU_CSSE-CSSE_Ensemble x OHT_JHU-nbxd	Partially Follow	Hierarchical structure (state, regional, national models) + adaptive MAE-weighting.
JHU_CSSE-CSSE_Ensemble x UM-DeepOutbreak	Follow + Innovate	Secondary model to predict error magnitudes & find quantiles of normalized residuals.
JHU_CSSE-CSSE_Ensemble x UMass-ar6_pooled	Follow	
JHU_CSSE-CSSE_Ensemble x UMass-gbqr	Follow	Combines predictions from 'adaptive' model trained on recent data & 'stable' model trained on longer history.
OHT_JHU-nbxd x UM-DeepOutbreak	Follow	
OHT_JHU-nbxd x UMass-ar6_pooled	Follow	Feature engineering + ensembling + variance-stabilizing transformation _ recursive forecasting.
OHT_JHU-nbxd x UMass-gbqr	Follow	Uses LightGBM predicts parameters of Gamma distribution.
UM-DeepOutbreak x UMass-ar6_pooled	Follow	
UM-DeepOutbreak x UMass-gbqr	Follow	
UMass-ar6_pooled x UMass-gbqr	Follow	LightGBM quantile regression on fourth-root transformed target.
DEEP-RESEARCH-CSTGT	Follow	Simplified static graph + synthetically generated policy feature.
DEEP-RESEARCH-MetaEnsembler	Follow	Meta-model to predict WIS.
DEEP-RESEARCH-FairnessAwareOptimization	Follow	Iterative re-weighting approximates composite fairness loss.
DEEP-RESEARCH-RegimeSwitchingDetection	Follow	
CO-SCIENTIST-STGNN-AgACI	Does not Follow	AR quantile regression model using LightGBM. Omits AgACI stage, replaces with simpler post-processing.
CO-SCIENTIST-MAPS	Partially Follow	3-stage ensemble: substitutes core models (GNN, TCN, GPR, MLP) with feature-engineered LightGBM proxies.
DEEP-RESEARCH-GenomiWastewater Fusion	Follow	Uses mock API calls.
DEEP-RESEARCH-AdversarialRecalibration	Follow + Innovate	Implements a post-hoc GAN structure. Composite loss function combining adversarial + pinball loss.
DEEP-RESEARCH-BehavioralSensing	Follow	Simulates external data.

Continued on next page

Supplementary Table 9 – continued from previous page

Method	Judgment	Notes
DEEP-RESEARCH-HierarchicalBayesianNODE	Follow	Three-level model: Negative Binomial observation layer, Neural ODE for jurisdiction-level dynamics, global hyper-priors for partial pooling.
CO-SCIENTIST-HGPC	Partially Follow	LightGBM quantile regression, uses feature engineering as proxy for complex stages.
DEEP-RESEARCH-PIDM	Follow	Implements conditional Denoising Diffusion Probabilistic Model (DDPM) with U-Net backbone, with loss function a weighted composite of standard diffusion loss and a physics-based regularization term derived from an SEIR-H model's outputs. Probabilistic forecasts generated by sampling from the learned reverse process.
CO-SCIENTIST-HQE	Partially Follow	Trains multiple base models, feeds their predictions into a meta-learner, then applies a conformal prediction step to adjust final quantiles. Uses multiple LightGBM models instead of suggested Prophet/TBATS for diversity, manually implements conformal prediction instead of using MAPIE.
DEEP-RESEARCH-CounterfactualSimulation	Follow + Innovate	Follows Monte Carlo structure: defines uncertain drivers with distributions, simulates N trajectories by applying sampled shocks to base median forecast, calculates empirical quantiles. Introduces Poisson noise on top of scenario-driven forecasts.
rep-OHT_JHU-nbxd	Follow	Implements TCN encoder and N-BEATS decoder architecture, including extension of parallel residual blocks to forecast mean and variance for Gamma distribution. The final forecast is generated as a median of an ensemble with varying lookback windows and initializations.
rep-CMU-TimeSeries	Follow	Implements a quantile autoregression model fit jointly across jurisdictions on smoothed, population-normalized data.
rep-UMass-ar6_pooled	Follow	Uses OLS on lagged, fourth-root transformed data to create a shared-coefficient AR model, then calculates separate variance parameters for each location based on residuals.
rep-UM-DeepOutbreak	Follow	Implements sequence-to-sequence model using a GRU and self-attention, with location embeddings. Uncertainty quantified post hoc using split conformal prediction on a recent time window.
rep-UMass-gbqr	Follow	Uses LightGBM with engineered features (lags for signal activity, location and population for location properties, date components for timing, and the horizon itself).
rep-JHU_CSSE-CSSE_Ensemble	Follow + Innovate	Implements three-tiered hierarchical ensemble, using Holt-Winters, regional LSTMs with Louvain grouping, and a national LSTM, combined with MAE-based weighting. Uses scaled residuals to create prediction intervals that adapt to the magnitude of the forecast to generate quantile predictions.

Continued on next page

Supplementary Table 9 – continued from previous page

Method	Judgment	Notes
rep-CMU-climate_baseline	Follow + Innovate	Averages geo-specific and geo-aggregated quantiles within a centered weekly window. Introduces a configurable 'smoothing_factor', which regularizes final predictions by pulling them towards zero.
rep-CEPH-Rtrend_covid	Follow	Lowpass filtering, daily interpolation, MCMC for Bayesian Rt estimation, and a renewal equation forecast. The Rt forecast correctly incorporates a sophisticated drift term that is modulated by the current incidence level.
retro_1	Follow	

Supplementary Table 10 | Full GIFT-Eval leaderboard (05/18/2025 snapshot)

Model	MASE	Type
Per-dataset	**0.671**	**tree-search**
TTM-R2-Finetuned	0.679	fine-tuned
timesfm_2_0_500m	0.680	pretrained
TabPFN-TS	0.692	pretrained
chronos_bolt_base	0.725	pretrained
Unified	**0.734**	**tree-search**
chronos_bolt_small	0.738	pretrained
PatchTST	0.762	deep-learning
TEMPO_ensemble	0.773	fine-tuned
VisionTS	0.775	pretrained
Chronos_large	0.781	pretrained
Moirai_large	0.785	pretrained
Chronos_base	0.786	pretrained
Chronos_small	0.800	pretrained
Moirai_base	0.809	pretrained
TFT	0.822	deep-learning
N-BEATS	0.842	deep-learning
Moirai_small	0.849	pretrained
TTM-R2-Zeroshot	0.915	pretrained
DLinear	0.952	deep-learning
Auto_Arima	0.964	statistical
TimesFM	0.967	pretrained
TTM-R1-Zeroshot	0.969	pretrained
Auto_Theta	0.978	statistical
TIDE	0.980	deep-learning
Seasonal_Naive	1.000	statistical
Timer	1.019	pretrained
Auto_ETS	1.088	statistical
Lag-Llama	1.102	pretrained
DeepAR	1.206	deep-learning
Naive	1.260	statistical
Crossformer	2.310	deep-learning

Supplementary Table 11 | Three example configurations from the final `unified solution`. Each dictionary defines a complete forecasting strategy discovered by the tree search, combining different components of the Iterative Decomposition Model. The validation process selects the best configuration for each dataset.

Unified Solution Example Configurations

```
config_list = [
  {
      'name': 'seasonal_naive_baseline',
      'description': 'Robust baseline...',
      'components': [{'type': 'base', 'method': 'seasonal_naive_adaptive'}],
      'transform_log': False, 'non_negative': False, 'version': 4,
  },
  {
      'name': 'additive_damped_linear_LogTransform',
      'description': 'General-purpose additive model...',
      'components': [
          {'type': 'base', 'method': 'median_all'},
       {'type': 'trend', 'method': 'polynomial', 'degree': 1, 'damping_factor': 0.90},
          {'type': 'seasonal', 'method': 'average', 'window_multiplier': 5.0},
      {'type': 'residual', 'method': 'median', 'window_size': 18, 'decay_factor': 0.90},
      ],
      'transform_log': True, 'non_negative': True, 'version': 4,
  },
  {
      'name': 'date_features_seasonal',
    'description': 'Robust additive model with key cyclical and datetime features...',
      'components': [
          {'type': 'base', 'method': 'median_all'},
          {'type': 'datetime', 'features': [
              ['dayofweek', 'hour'], 'month', 'is_month_start', 'weekofyear',
              'is_weekend', 'is_quarter_start',
              {'name': '_is_holiday_flag',
               'country_codes': ['US', 'DE', 'CN', 'GB', 'CA', 'AU']}
          ]},
          {'type': 'seasonal', 'method': 'average', 'window_multiplier': 4.0},
      {'type': 'residual', 'method': 'median', 'window_size': 14, 'decay_factor': 0.92},
      ],
      'transform_log': False, 'non_negative': False, 'version': 4,
  },
  % ... other configurations can be added here ...
]
```

Supplementary Table 12 | Prompt for Gemini Deep Research to generate ideas to integrate single-cell batch effects.

Prompt for Gemini Deep Research.

I am developing new methods for winning single-cell batch integration competitions, as proposed by the Kaggle and extensively researched in the single-cell genomics community.

Briefly: Modelers are asked to develop a function, `eliminate_batch_effect_fn`, that transforms raw gene expression count data from multiple batches into a low-dimensional embedding or feature matrix. This transformed output should effectively remove technical variation (batch effects) while rigorously preserving biological information (e.g., cell type identity). The performance of these methods is evaluated against a suite of metrics that quantify both batch mixing and biological conservation.

The key problem is to develop a method that takes an `AnnData` object of raw gene expression counts with batch labels and returns an AnnData object with a batch-integrated low-dimensional embedding in the `.obsm['X_emb']` field. The method must excel across a diverse set of evaluation metrics, including ASW Batch, ASW Label, ARI, NMI, Graph Connectivity, Isolated Labels ASW, Isolated Labels F1, kBET, iLISI, cLISI, PCR, and Cell Cycle Conservation Score, aiming to maximize their average.

The following principles should be obeyed when choosing models:
* **Batch Effect Removal**: Prioritize techniques that explicitly model and mitigate batch-specific variations without collapsing biological signal.
* **Biological Conservation**: Ensure the integrated representation retains and accurately reflects genuine biological differences, particularly cell type distinctions, as measured by clustering and silhouette metrics.
* **Scalability and Efficiency**: Given the large dataset sizes (e.g., $329,762$ cells $\times$ $2,000$ genes), models must be computationally efficient and avoid out-of-memory errors.
* **Constraint Adherence**: The implementation must strictly avoid using cell_type information during integration and should primarily leverage `scanpy`, `sklearn`, `numpy`, `scipy`, `tensorflow`, `torch`, `jax`, or equivalent native implementations rather than specialized single-cell packages.

This task aims to develop a SUPERHUMAN METHOD for solving this problem.

Please give me 10 highly novel and creative ideas with detailed implementation notes for the set of methods I should explore for solving this task. I aim to create the best method for solving this problem, preferably creating the best ever method.

Supplementary Table 13 | Prompt for formatting Deep Research ideas into a structure similar to baseline method descriptions.

Prompt for formatting Deep Research ideas.

```
Structure the given idea into the following format:

<description>
Your description about the method goes here.
</description>

<steps>
Your list of steps to implement the method goes here.
</steps>

<notes>
Strengths and weaknesses of the idea goes here.
</notes>
```

Supplementary Table 14 | Prompt for guiding tree search to generate hybrid strategies.

Prompt for guiding tree search to generate hybrid strategies.

We have up until now done experiments with two major types of codes, that are described in detail below. PLEASE CREATE AN ALGORITHM THAT USES THE BEST PARTS OF BOTH STRATEGIES TO CREATE A HYBRID STRATEGY THAT IS TRULY WONDERFUL AND SCORES HIGHER THAN EITHER OF THE INDIVIDUAL STRATEGIES.

[Output from Supplementary Table 6.]

Supplementary Table 15 | Prompt for summarizing existing batch integration methods. The prompt instructs the LLM to write a brief description and step. This generated summary then serves as part of an explicit instruction for tree search to create optimized version of the existing batch integration method. Example output in shown in Supplementary Table 16.

```
Prompt for summarizing differences between two baseline methods.

Given the following paper, please identify the main method being proposed. Then write a very
short method description. This method description will be used to reproduce the method. DO
NOT mention the algorithm by name.

Your output must follow this format:
<description>
Your very short description goes here.
</description>

<steps>
Your short list of steps goes here.
</steps>
```

Supplementary Table 16 | Example of existing method description (BBKNN) given to tree search. Tree search used this short method description to further optimize the method.

Example of existing method description (BBKNN) given to tree search.

<description>
This method performs batch correction by modifying the neighborhood graph construction step. For each cell, its nearest neighbors are identified independently within each batch, rather than across the entire combined dataset. The resulting batch-specific neighbor lists for each cell are then merged to create a single, integrated graph. This approach assumes that shared cell types exist across batches and that biological differences are greater than technical batch effects.
</description>

<steps>

1. For each cell, iterate through every batch in the dataset.
2. Find the k-nearest neighbors for the cell from within the current batch, based on a given distance metric (e.g., Euclidean distance in PCA space).
3. After iterating through all batches, merge the identified neighbor sets for the cell into a single neighborhood.
4. Repeat for all cells to construct a batch-corrected neighborhood graph.

</steps>