IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

AI-Powered Storage System Classifier: Intelligent File Tiering Using Access Patterns

Nooka Harshavardhan[1], T. V. Sai Prudhvi[2]

Department of C omputer Science and Engineering, SRM Institute of Science and Technology, Vadapalani, Chennai, India[1,2]

Abstract: As digital systems generate and retain increasing volumes of data, selecting an appropriate storage tier becomes important for balancing access speed, storage cost, and resource utilization. This paper presents an AI-powered storage system classifier that categorizes files into Hot, Warm, and Cold storage tiers based on observed file metadata and access behaviour. The proposed system collects metadata such as access frequency, time since last access, file age, file size, and recent access trends. These features are supplied to a Random Forest classification model, which predicts the most suitable storage tier. Frequently accessed files are assigned to Hot storage, files accessed occasionally are assigned to Warm storage, and rarely accessed files are assigned to Cold storage. The design includes a monitoring process that periodically updates file activity and supports reclassification when usage patterns change. Python, Pandas, NumPy, scikit-learn, Streamlit, and SQLite are proposed for the implementation, with a local filesystem or cloud object storage such as AWS S3 serving as the storage backend. This paper describes the system design and evaluation plan; measured accuracy and performance results are not claimed until implementation and testing are completed.

Keywords— storage tiering, machine learning, Random Forest, file classification, access patterns, Hot storage, Warm storage, Cold storage, intelligent storage management

I. INTRODUCTION

Modern organizations store documents, images, videos, backups, application data, and other digital files. These files do not all have the same access requirements. Some are opened repeatedly and need fast access, while others are accessed only occasionally or retained mainly for long-term reference. Keeping every file on the same high-performance storage can increase cost and waste resources.

Storage tiering addresses this problem by placing data in different storage classes according to performance and access requirements. A basic rule-based system can use fixed thresholds, but static rules may not adapt well when file usage changes over time. A file that was frequently accessed last month may become inactive, while an older file may suddenly become important.

The proposed AI-Powered Storage System Classifier uses machine learning to estimate the suitable tier from file metadata and access history. It focuses on three classes: Hot, Warm, and Cold. The classifier is intended to support informed placement decisions and periodic reclassification, rather than treating a file's initial classification as permanent.

The main objectives are to collect useful file-level features, train a supervised classification model, predict a storage tier, and provide a simple interface to inspect the classification. The design also separates prediction from physical migration so that tier assignment can be tested safely before automated movement is enabled.

An important consideration in storage management is that access patterns are uneven. A small group of files may account for most read operations, while a large amount of stored data may remain untouched for long periods. Treating every file identically can therefore lead to inefficient use of high-performance storage. A tiered design attempts to align the cost and performance of storage resources with the expected demand for each file.

The three-tier model provides a practical starting point for this project. Hot storage is intended for active files that are accessed frequently or have been accessed very recently. Warm storage is intended for files that are still useful but are accessed less often. Cold storage is intended for files that are rarely accessed and can tolerate a longer retrieval time. The boundaries between these categories should be defined by an explicit policy and validated against actual usage data.

The proposed classifier is decision-support software. It analyzes historical metadata and recommends a tier; it does not independently determine the business importance of a file. For example, an old legal record may be rarely accessed but still require rapid retrieval for compliance reasons. Such exceptions can be handled through user-defined rules or

 This work is licensed under a Creative Commons Attribution 4.0 International License

protected-file flags that override a model recommendation. This separation between learned predictions and administrative policy is important for a dependable storage-management system.

The expected contribution is an integrated prototype that demonstrates how machine learning can support storage-tier decisions. The work focuses on file-level classification, explainable input features, and a monitoring loop. It does not claim to introduce a new machine-learning algorithm or a production-grade cloud migration service.

Another design challenge is deciding what the system should do when the available metadata is incomplete. A file that has just been added may not have an access history, and some file systems may not expose a dependable last-access timestamp. The classifier should not interpret missing values as zero activity without checking why the value is missing. The prototype can use a documented default tier for new files and update the prediction after enough access events have been collected.

Storage tiering also involves a trade-off between access performance and total cost. Moving a file to a lower-cost tier can increase retrieval time or incur a retrieval fee, while keeping every file in a high-performance tier can increase storage expenditure. Therefore, the best tier is not necessarily the one with the lowest storage price. The system should make its classification policy explicit and allow the administrator to consider retrieval requirements before approving a tier change. To keep the project manageable, the initial implementation can use a local directory and a small metadata database. This allows the team to demonstrate feature extraction, model prediction, tier visualization, and controlled movement without requiring a cloud account. Once the local workflow is stable, the same classification interface can be connected to a cloud backend through a separate storage adapter.

II. RELATED WORK

A. Storage Tiering

Storage tiering organizes data into classes that differ in access performance, availability, and cost. Frequently accessed data is generally placed on faster storage, while infrequently accessed data can be placed on lower-cost storage. Cloud object-storage platforms also provide storage classes intended for different access patterns. The proposed project applies this general principle at the file-classification level.

B. Metadata-Based Classification

File metadata provides measurable signals about likely future access. Access frequency indicates how often a file is used, while the time since last access helps distinguish active files from inactive ones. File age and size provide additional context, and recent access trends can reveal changes that a lifetime count may hide. These features form the input to the proposed classifier.

C. Machine Learning for Classification

Supervised machine learning can learn relationships between input features and known class labels. Random Forest is an ensemble method that combines predictions from multiple decision trees. It can model non-linear relationships and handle a mixture of numerical features, making it a reasonable baseline for this classification task. Its performance must still be measured on representative and correctly labelled data.

D. Monitoring and Reclassification

Access behaviour is not fixed. A practical tiering system therefore needs to monitor file activity and revisit earlier decisions. Periodic reclassification can identify files whose usage patterns have changed. Migration policies must also consider transfer time, temporary storage requirements, service interruption, and the cost of moving data.

Storage systems commonly distinguish between media and service classes according to their performance characteristics. Solid-state storage is typically selected for workloads that benefit from low latency, while lower-cost capacity-oriented storage can be suitable for infrequently accessed information. Cloud object-storage services expose storage classes with different retrieval and pricing characteristics. These existing approaches establish the practical motivation for tiering, although the exact names, minimum storage durations, retrieval charges, and availability guarantees depend on the provider.

Rule-based tiering is a straightforward approach in which an administrator defines conditions such as access count or elapsed time since the last read. It is transparent and easy to implement, but fixed thresholds may not capture combinations of signals. A file with moderate recent activity and a high lifetime access count may need a different decision from a file with the same lifetime count but no recent activity. A supervised model can learn such combinations when representative labelled examples are available.

 This work is licensed under a Creative Commons Attribution 4.0 International License

Monitoring is a complementary requirement rather than a separate optional feature. A classification made once can become outdated as users, applications, and workloads change. Periodic observation and reclassification are therefore necessary to keep placement decisions aligned with current usage. The proposed design brings these ideas together in a manageable student project prototype.

III. SYSTEM ARCHITECTURE

The system is organized as a pipeline that converts file activity information into a storage-tier recommendation. The main modules are file input, metadata collection, feature extraction, Random Forest classification, tier decision, and monitoring. The storage layer may be implemented using local directories or a cloud object-storage backend.

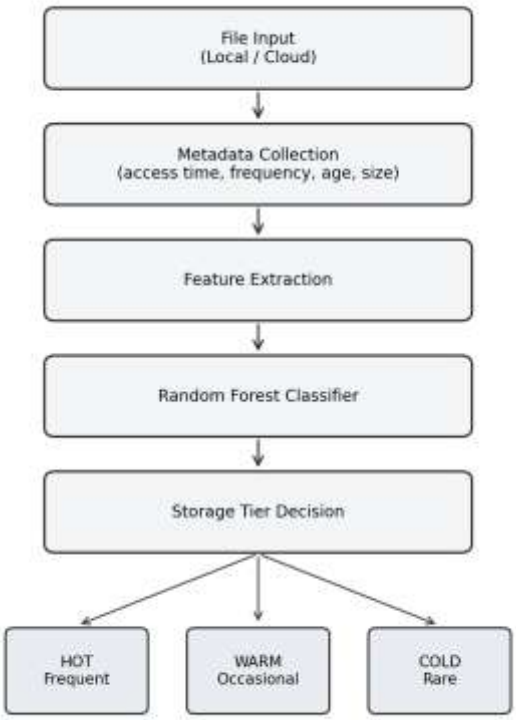


Fig. 1. Proposed architecture of the AI-powered storage system classifier. File metadata is transformed into features, classified into a tier, and monitored for future reclassification.

A. File Input and Metadata Collection

The system begins with files available in a local directory or a configured storage backend. For each file, the metadata collector records the attributes required for classification. The proposed attributes include access frequency, last-access timestamp, file creation or modification age, file size, and recent access trends. Where the operating system or storage provider does not expose reliable access timestamps, the system should maintain its own access log.

B. Feature Extraction

Raw metadata is converted into numerical features suitable for the model. Access frequency can be represented as a count over a defined observation window. Time since last access can be calculated from the current timestamp and the recorded last-access time. File age is derived from the file's creation or first-seen date. File size is represented in a consistent unit, and recent access trends can be calculated from counts in successive time windows. The observation window and feature definitions must remain consistent between training and prediction.

C. Random Forest Classification

The extracted feature vector is passed to a trained Random Forest model implemented with scikit-learn. Each decision tree predicts a storage class, and the ensemble combines the tree outputs to produce the final class prediction. The model is trained using examples labelled as Hot, Warm, or Cold. The quality of the model depends on the representativeness of the training data and the consistency of the labelling policy.

 This work is licensed under a Creative Commons Attribution 4.0 International License

D. Storage Tier Assignment

The predicted label is mapped to a storage tier. Hot represents files with frequent or recent access and is associated with faster storage. Warm represents files with occasional access and a moderate performance requirement. Cold represents files with rare access and a stronger emphasis on lower storage cost. The project should define the physical meaning of each tier for the selected backend; the labels alone do not guarantee a particular latency or price.

E. Monitoring and Reclassification

The monitoring module records subsequent access events and refreshes the metadata features. At a configured interval, the classifier can reassess files and identify tier changes. Before enabling automatic migration, the system should support a recommendation or preview mode. This helps verify that classification decisions are sensible and reduces the risk of unnecessary file movement.

The architecture separates data collection, prediction, and storage operations so that each responsibility can be tested independently. The metadata collector is responsible for identifying files and recording permitted attributes. The feature extractor converts these attributes into a consistent representation. The classifier predicts one of the three target labels, and the tier-decision component translates that label into a recommendation for the configured storage backend.
Access frequency should be measured over a defined observation window, for example the number of recorded reads during the previous thirty days. Time since last access should be computed consistently using the same clock and time zone. File age and file size add context, but should not dominate the decision without validation. If access events are collected by the application, the logger should record the event type and timestamp while avoiding unnecessary capture of file contents.

The tier-decision module should initially operate in recommendation mode. It can display the predicted class and the reason-related feature values without moving data. After the prediction process has been validated, an administrator may enable controlled migration. The migration controller should verify that the destination is available, copy the file, compare size or checksum, update the metadata record, and only then remove the old copy if deletion is explicitly permitted. This staged approach reduces the risk of data loss.

Storage lifecycle management is another related concept. Lifecycle policies automatically transition objects between storage classes after a specified period or based on predefined conditions. Such policies are useful because they reduce manual administration, but they generally depend on rules selected by an administrator. The proposed classifier differs in emphasis by using several observed file attributes together to estimate a tier. A lifecycle policy and a machine-learning classifier could also complement one another: the model can recommend a class, while policy rules enforce minimum retention, legal holds, and other operational constraints.

Feature engineering is central to tabular machine-learning tasks. The quality of the classifier depends not only on the algorithm but also on whether the input variables represent the behaviour that matters. For storage tiering, a count of accesses over a recent period can be more informative than a lifetime count because it reflects current demand. A recency feature can distinguish files with similar counts but different last-access times. The project should compare candidate features and avoid including identifiers that merely allow the model to memorize individual files.

IV. PROPOSED METHODOLOGY

A. Data Collection and Labelling

A dataset is required with one record per file or file-observation period. Each record should contain the selected metadata features and a target label representing the desired storage tier. Labels may be assigned using a documented access-policy rule or curated usage scenarios. If labels are generated from thresholds, those thresholds must be stated clearly because the model will learn the policy represented by those labels.

B. Preprocessing

The preprocessing stage checks missing timestamps, invalid file sizes, duplicate records, and inconsistent units. Time-based fields are converted into durations or counts. Missing values are handled using a documented strategy, and the same preprocessing steps are applied during training and inference. The dataset is divided into training and test subsets so that evaluation is performed on records not used to fit the model.

C. Model Training

A Random Forest classifier is trained on the labelled training subset. Model settings such as the number of trees, maximum tree depth, and minimum samples per leaf can be selected using validation data. Class distribution should be

inspected because an imbalance between Hot, Warm, and Cold examples can make overall accuracy misleading. The trained model and preprocessing configuration are saved together for repeatable predictions.

D. Prediction and Tier Mapping

For a new or updated file record, the system applies the saved preprocessing pipeline, obtains the feature vector, and requests a prediction from the model. The output class is displayed with the file name and relevant metadata. The user interface can also show a confidence or class-probability estimate where supported by the model, while making clear that this is not a guarantee of future access behaviour.

E. Storage Integration

The prototype can begin with local folders representing Hot, Warm, and Cold tiers. A later integration may map these tiers to storage classes in a cloud object-storage service such as AWS S3. Actual movement should be implemented as a separate, controlled operation with permission checks, error handling, and a record of the source and destination. The initial project design does not assume that cloud migration has already been implemented.

The observation window is a key methodological choice. If the window is too short, a temporary burst of activity may incorrectly move a file into Hot storage. If it is too long, a recent change in usage may be hidden by older activity. The selected window should therefore be documented and, where possible, compared with alternative windows during validation. Multiple windows, such as seven-day and thirty-day access counts, may provide additional information about short-term and longer-term behaviour.

Data splitting must avoid leakage between training and test records. If multiple observations belong to the same file, randomly placing some observations in training and others in testing can make evaluation overly optimistic. A file-level split or a time-based split is preferable when the dataset permits it. A time-based test is particularly useful because it approximates the real task of predicting a suitable tier from earlier activity.

Model training should begin with a simple baseline, such as a rule-based threshold classifier, so that the Random Forest can be compared against a transparent reference. The model should be trained only on the training subset. Hyperparameter selection should use validation data or cross-validation, and the final test set should be used only for the final performance report. The class labels and thresholds used to create them must be documented.

After training, the complete prediction pipeline should be saved, including feature order, preprocessing rules, and model version. During inference, the system must supply features in exactly the same order and units used during training. This avoids errors caused by a model receiving, for example, file size in bytes during training and megabytes during prediction. Every prediction should be associated with a timestamp and model version. Keeping a prediction history makes it possible to investigate why a file changed tiers and to compare recommendations over time. It also provides a useful basis for debugging and for later evaluation of whether the model's decisions were stable.

Class labels should be assigned according to a written policy. For example, the project may define Hot, Warm, and Cold using ranges of recent access frequency and elapsed time since last access. The exact thresholds should be selected based on the dataset and project requirements rather than presented as universal values. If labels are assigned manually, the labelling guide should be shared with all labelers and ambiguous records should be reviewed consistently.

Feature preparation should include checks for extreme values. A file may have a very large size, an unusually high access count, or an invalid timestamp due to a clock or logging error. Such records should be inspected before training. Numerical features may be kept in their natural units for Random Forest, but consistent units and sensible missing-value handling are still required. Any filtering or correction should be recorded so the experiment can be reproduced.

After model fitting, predictions should be checked against a small set of known examples before the model is connected to the dashboard. This manual verification can catch reversed labels, inconsistent feature ordering, and errors in the preprocessing code. The team should also test boundary cases, including a new file with no access history, a file with frequent recent reads, and a file that was active in the past but has become inactive.

An operational test should simulate a change in access pattern. For example, a file may initially be classified as Cold and then receive repeated accesses during a later observation window. The monitoring process should update the metadata and allow the model to reconsider the tier. The purpose of this test is to verify that reclassification responds to changing activity rather than permanently retaining the first prediction.

V. TECHNOLOGY COMPONENTS

A. Python and Data Processing Libraries

Python is the proposed implementation language. Pandas can organize file metadata in tabular form, while NumPy can support numerical transformations and feature preparation. These libraries provide the data-processing layer used before model training and prediction.

B. Scikit-learn Random Forest

Scikit-learn provides the RandomForestClassifier implementation used for supervised learning. It supports model fitting, prediction, and evaluation through standard machine-learning workflows. The final model configuration should be selected through validation rather than assumed in advance.

C. Streamlit and SQLite

Streamlit is proposed for a lightweight dashboard where users can inspect files, view predicted tiers, and review classification summaries. SQLite can store file metadata, access logs, prediction history, and tier-change records in a local database. A local filesystem or AWS S3 can act as the storage backend depending on the implementation scope.

The project can be developed in Python because it provides libraries for data handling, machine learning, database access, and dashboard development. Pandas is suitable for loading metadata records, inspecting missing values, grouping access events, and preparing tabular features. NumPy supports numerical operations used in feature preparation. Keeping these operations in a repeatable preprocessing pipeline helps ensure that training and inference use the same transformations.
Scikit-learn provides the RandomForestClassifier and evaluation utilities such as classification reports and confusion matrices. The project can use a train-test split for an initial experiment and cross-validation for model selection. The trained model may be persisted with an appropriate serialization method, and the implementation should record the library version to support reproducibility. Serialized model files should be treated as trusted application artifacts and not loaded from unknown sources.

SQLite is appropriate for a small prototype because it stores structured records in a single local database and does not require a separate database server. Tables can represent file metadata, access events, prediction history, and migration activity. Indexes on file identifiers and timestamps can improve queries as the event history grows. For a larger deployment, the database layer could later be replaced with a server-based database.

Streamlit can provide a simple interface with a file inventory, tier counts, filters, and a view of individual file metadata. The dashboard should distinguish between a predicted tier and the file's current physical location. It should also show when the record was last refreshed, because an old prediction may no longer reflect current access behaviour.

A local filesystem is a suitable first backend for demonstrating the concept. Separate directories can represent Hot, Warm, and Cold tiers, allowing the migration workflow to be tested without cloud charges. If AWS S3 is later selected, the mapping from project tiers to provider storage classes must account for retrieval latency, retrieval charges, minimum storage duration, and lifecycle rules. These provider-specific properties should be checked against current documentation before deployment.

The application can be organized into separate modules for scanning files, logging access events, calculating features, loading the trained model, and displaying results. This modular structure reduces coupling and makes it easier to test individual components. For example, the feature-extraction module can be tested using fixed metadata records without requiring the dashboard or storage backend to be running.

The metadata database should store the time at which each record was collected and the source of the access information. A file inventory table can store the file identifier, path, size, creation or first-seen time, and current tier. An access-event table can store the file identifier, event type, and timestamp. A prediction table can store the predicted class, model version, and prediction time. These records provide a basic audit trail for the prototype.

The dashboard should present information in a way that supports review rather than hiding the model's uncertainty. Useful views include the number of files in each tier, a searchable file list, recent tier changes, and a detail panel for a selected file. If the classifier provides class probabilities, they can be displayed as model confidence estimates, with a note that they do not guarantee future access behaviour.

Cloud support should be implemented through a backend interface so that the classifier does not depend directly on a particular provider. A local adapter can move files between directories, while a cloud adapter can call the provider's

object-storage API. The interface should define common operations such as listing objects, reading metadata, copying an object, verifying a copy, and updating the object's storage class. This separation will make future integrations easier to maintain.

VI. EVALUATION FRAMEWORK

The project should be evaluated at both the model level and the system level. No measured accuracy, latency, or cost-saving result is asserted in this paper because finalized dataset details and experimental results have not been provided. The following framework identifies the measurements to report after implementation.

Evaluation area	Purpose	Metrics / observations
Classification	Measure how well files are assigned to Hot, Warm, and Cold	Accuracy, precision, recall, F1-score, confusion matrix
Class balance	Identify weak performance on less common tiers	Per-class support and macro F1-score
Prediction time	Measure inference overhead for file classification	Average and percentile prediction time
Reclassification	Check whether changing access patterns are detected	Tier-change correctness and frequency
System operation	Assess metadata collection and storage workflow	Metadata collection time, migration success/failure, error logs

The test set should include files with frequent, occasional, and rare access patterns, along with examples whose usage changes over time. Evaluation should report per-class metrics as well as overall accuracy. System tests should separately measure metadata collection, prediction time, and any file-migration operation. Results should be reported only after the corresponding tests have been run.

Classification accuracy is the proportion of test records assigned to the correct class, but it should not be used alone. If most files in the dataset are Cold, a model that predicts Cold too often may obtain a high accuracy while performing poorly on Hot files. Precision measures how many files predicted as a tier truly belong to that tier, while recall measures how many of the actual files in that tier are correctly identified. The F1-score balances precision and recall.

A confusion matrix is useful because it shows which tier pairs are being confused. Confusing Warm with Cold may have a different operational impact from confusing Hot with Cold. The evaluation should therefore discuss the consequences of each error type, not just provide one aggregate score. Macro-averaged metrics give equal weight to each class and are useful when the class distribution is uneven.

The test dataset should represent realistic access patterns. It should include consistently active files, files with occasional access, long-unused files, and files whose usage changes after a period of inactivity. If the dataset is generated synthetically, the generation rules and assumptions must be stated. Synthetic results can demonstrate the pipeline, but they do not prove performance on real organizational workloads.

System evaluation should measure the time needed to scan metadata, construct features, and generate predictions for a batch of files. If migration is implemented, tests should record successful transfers, failed transfers, checksum mismatches, and recovery behaviour. Cost savings should not be claimed unless the project measures or calculates the relevant storage and retrieval costs using clearly stated provider prices and workload assumptions.

The final report should include the dataset size, class distribution, train-validation-test split, model parameters, software versions, and evaluation date. Results should be reproducible and should distinguish between model metrics and operational measurements. Until those experiments are completed, the paper should describe the evaluation plan rather than present hypothetical numbers as actual performance.

An evaluation should first establish a baseline. A threshold-based policy can be implemented using the same access-frequency and recency features. The Random Forest model can then be compared with that baseline using the same test records. If the model does not improve the relevant metrics or produces decisions that are difficult to justify, the simpler rule-based method may be preferable for the prototype.

Evaluation should also consider the cost of different classification errors. Predicting Cold for a file that is frequently used may cause poor user experience or unexpected retrieval charges. Predicting Hot for a file that is rarely used may waste

 This work is licensed under a Creative Commons Attribution 4.0 International License

storage budget but is less likely to affect access speed. The project can describe these impacts and, if suitable data is available, use a weighted error analysis to reflect the different consequences.

To test stability, the team can run the classifier at multiple observation dates and compare how often files change tiers. Excessive movement may indicate noisy features, unstable labels, or thresholds that are too sensitive. A minimum residence period, smoothing of access counts, or a confidence requirement may reduce unnecessary transitions. Any such change should be evaluated rather than assumed to improve the system.

Results should be reported with enough detail for another student to reproduce the experiment. At minimum, the report should identify the source and size of the dataset, the target-label policy, the feature list, the split method, model settings, software versions, and the metrics obtained. The final paper can then replace the current evaluation plan with measured results from the completed implementation.

VII. ILLUSTRATIVE CLASSIFICATION EXAMPLE

Table II shows hypothetical examples to explain the intended tier mapping. These are illustrative records, not measured outputs from a completed model. The final class prediction will be generated by the trained Random Forest model using the dataset and feature definitions adopted during implementation.

File	Access pattern	Last accessed	Size	Illustrative tier
report.docx	Frequent	Today	2 MB	Hot
lecture.pdf	Occasional	12 days ago	18 MB	Warm
archive.zip	Rare	14 months ago	1.2 GB	Cold

Table II. Hypothetical examples showing the intended interpretation of the three storage tiers.

Consider three example files in a user's storage directory. A report that is opened several times each day and was accessed today has a high recent access count and a very small time-since-last-access value. The model may therefore assign it to Hot storage. A lecture document opened a few times during the month may be assigned to Warm storage. A compressed archive that has not been opened for more than a year may be assigned to Cold storage, provided that no administrative rule requires it to remain on faster storage.

The examples illustrate the relationship between the observed features and the intended meaning of the tier labels. They are not fixed rules that the model must follow in every case. The Random Forest learns from labelled training examples, so a file can receive a different prediction if the access-frequency window, training distribution, or labelling policy changes. The project should retain the feature values alongside each prediction so that users can understand the context of the recommendation.

A useful interface can show the file name, current location, predicted tier, access count, last-access date, and prediction time in a table. Users should be able to filter by tier and inspect files whose predicted tier differs from their current location. A preview of proposed movements can help administrators identify unexpected decisions before enabling migration.

An important edge case is a file that is rarely accessed but must remain immediately available, such as a critical configuration file or a time-sensitive operational record. The classifier may predict Cold from its access history, but a protected-file policy should override that recommendation. Another edge case is a newly created file with no access history. The system should assign a documented default or use a temporary Warm tier until sufficient observations are collected, rather than treating missing history as evidence that the file is unimportant.

The classifier can be demonstrated with a small scenario in which a user stores a project report, lecture notes, and an old backup archive. The report is edited and opened regularly, so the access logger records repeated activity. The lecture notes are opened occasionally during the semester, while the backup archive is retained for reference and is almost never opened. The system calculates the features for each file and submits them to the trained model. The resulting labels are shown in the dashboard as recommendations.

The example also demonstrates why multiple features are useful. Two files may have the same last-access date but different access frequencies. One may have been opened many times during a short period, while the other was opened only once. Similarly, two files may have the same access count but very different recency. Considering these signals together gives the model more information than using file age or file size alone.

A prediction should be treated as a point-in-time recommendation. If the lecture notes become part of an active course project, their access frequency may increase and a later prediction may move them to Hot storage. If the report is archived after project completion, its activity may decline and a later prediction may move it to Warm or Cold storage. This illustrates the purpose of monitoring and reclassification.

For a fair demonstration, the team should show the input features, the predicted tier, and the reason the example is included. If the records are synthetic, the interface or report should identify them as sample data. A demonstration using a few hand-selected files is useful for showing functionality, but it should not be confused with a statistical evaluation of model performance.

VIII. LIMITATIONS AND ETHICAL CONSIDERATIONS

The classifier predicts a tier from historical metadata; it cannot know with certainty when a file will be accessed next. Sudden changes in user behaviour can make earlier predictions less suitable. Access timestamps may also be unavailable, inaccurate, or affected by file-copy operations. These limitations should be documented and addressed through reliable logging and periodic reassessment.

A poor labelling policy can teach the model undesirable placement decisions. The project should define Hot, Warm, and Cold consistently and review class balance before training. The system should also avoid deleting data as part of tiering. Migration must preserve file contents, permissions, and metadata, and should include failure recovery and audit logs.
File names and access histories can reveal sensitive information about users. The implementation should collect only necessary metadata, restrict access to logs, protect credentials, and define a retention policy. If cloud storage is used, access permissions and provider-specific security controls must be configured carefully.

Historical access is only an imperfect estimate of future demand. A file that has not been opened for months may become important tomorrow, and an active file may suddenly become obsolete. The classifier should therefore be treated as a recommendation mechanism with uncertainty, not as an infallible controller. A user or administrator should be able to override a prediction when business requirements demand it.

Metadata quality is another limitation. Some operating systems do not update last-access timestamps reliably, and backup, synchronization, or antivirus processes may change file metadata without representing a user's actual interest. If access frequency is based on application logs, events may be missed when users access files outside the monitored application. The implementation must explain how access is observed and what kinds of activity are counted.

Migration can create operational risks even when the classification is correct. Moving a file may temporarily increase latency, interrupt an application, or fail because of permissions or network problems. The system should use copy-and-verify operations, preserve access permissions, maintain a migration log, and provide a recovery path. It should not delete the original file until the destination copy has been verified and the configured policy authorizes the change.

Privacy and security must be considered because file names, paths, and access histories can reveal personal or organizational activity. The prototype should collect only the metadata needed for classification, restrict access to its database and dashboard, and avoid collecting file contents. Credentials for cloud services should be stored securely and excluded from source-code repositories. Data-retention rules should define how long access logs and prediction history are kept.

Finally, the training labels may encode the assumptions of the person who created them. If the labelling policy is poorly designed, the model can reproduce those weaknesses at scale. The policy should be reviewed, class distributions should be examined, and the system should retain a manual override for exceptional files.

The proposed system also has limits related to workload diversity. Access patterns for office documents may differ from those for media archives, databases, virtual-machine images, or application logs. A model trained on one type of workload may not generalize to another. If the project is later used in a different environment, its feature distributions and performance should be checked before deployment.

Storage providers may apply different rules to their storage classes. Some classes have minimum storage durations, minimum object sizes, retrieval fees, or different availability characteristics. A project-level label such as Cold should therefore not be assumed to map identically to every provider's archive class. The backend configuration should document the actual class selected and the expected retrieval implications.

Users should be informed when files are being monitored and what metadata is collected. The system should not inspect document contents when file-level metadata is sufficient for the classification task. Administrative controls should restrict who can view access logs and initiate migration. These practices reduce unnecessary exposure of potentially sensitive activity information.

Before automatic movement is enabled, the implementation should provide a dry-run report, a confirmation step, and a way to restore the previous location. Important files may be exempted from automatic movement through an allowlist or protected flag. These safeguards recognize that a prediction based on historical behaviour cannot capture every legal, operational, or personal requirement.

IX. FUTURE WORK

Future improvements may include learning from longer access histories, adding more detailed temporal features, comparing Random Forest with alternative classification algorithms, and tuning model parameters using validation data. A larger and more diverse dataset would help assess whether the model generalizes across different workloads.
The system may also be extended with automated migration policies, cloud storage integration, cost-aware tier selection, and a dashboard showing tier distribution and access trends. Before full automation, the migration workflow should be tested with dry-run recommendations and verified recovery procedures.

One direction for future work is to improve the feature set. Instead of relying only on a single access count, the classifier could use multiple observation windows, read-versus-write counts, access recency, and the regularity of usage. Time-of-day or day-of-week patterns may be useful for some workloads, but they should be added only when the data supports them and when collecting them is appropriate.

Alternative models can be compared with Random Forest, including Decision Tree, Gradient Boosting, and other suitable classifiers. A simpler model may be preferable if it provides similar performance with easier interpretation or lower inference overhead. The comparison should use the same dataset splits and evaluation metrics to ensure a fair assessment.
A further extension is cost-aware tier selection. In addition to predicting access frequency, the system could estimate the expected cost of keeping a file in each tier by considering storage charges, retrieval charges, request charges, and migration overhead. Such a model would need workload assumptions and current provider pricing. The lowest storage price does not always produce the lowest total cost when retrieval is frequent.

Cloud integration can be expanded after the local prototype is stable. A migration service could map the project tiers to storage classes supported by a selected provider and apply lifecycle policies. It should include access-control checks, retry handling, audit logs, and safeguards against accidental deletion. A dry-run mode would allow administrators to inspect proposed changes before applying them.

Longer-term evaluation should use real access traces from multiple workload types, subject to permission and privacy constraints. It should examine model drift, tier changes, migration stability, and the effect of prediction errors on users. These improvements would help determine whether the proposed approach is useful beyond a controlled academic prototype.

Future work can examine online or incremental learning, where the model is periodically updated as new labelled observations become available. This approach may help the classifier adapt to changing workloads, but it also introduces a need to monitor model drift and prevent noisy recent events from degrading the model. Retraining should therefore be controlled and validated before a new model replaces the existing version.

Another extension is to include a cost and performance policy layer after classification. The model could estimate the likely access category, while a separate policy engine considers retrieval-time requirements, storage charges, and business constraints. This separation may be safer than asking a single classifier to learn every operational rule from historical data.

The dashboard could be expanded to include time-series charts showing access frequency and tier changes. Administrators could compare current and recommended placement, review files that have changed classification, and approve or reject proposed movements. These features would make the prototype more useful for demonstrating explainability and operational control.

Further work may include testing on larger datasets and comparing performance across different file types and usage scenarios. A controlled pilot could measure whether the proposed recommendations reduce storage cost without causing

unacceptable retrieval delays. Such a study would need to define the workload, baseline placement policy, provider pricing, and success criteria before collecting results.

X. CONCLUSION

This paper presented the design of an AI-powered storage system classifier that uses file metadata and access patterns to recommend Hot, Warm, or Cold storage tiers. The proposed pipeline collects file information, extracts features such as access frequency and time since last access, applies a Random Forest classifier, and maps the prediction to a storage tier. A monitoring component supports periodic reclassification as usage patterns change.

The design combines Python-based data processing, scikit-learn model training, Streamlit visualization, SQLite metadata storage, and a local or cloud storage backend. The project is intended to improve storage organization by matching file placement to observed usage. Its effectiveness must be established through a labelled dataset and measured evaluation; no experimental performance claim is made at this design stage.

The proposed design addresses a practical storage-management problem: files have different access patterns, but storing all files in the same class may not provide an appropriate balance between performance and cost. By using access frequency, time since last access, file age, file size, and related metadata, the system attempts to classify files according to their observed usage.

The architecture separates metadata collection, feature extraction, Random Forest prediction, tier assignment, and monitoring. This modular structure makes it possible to test the classifier independently from the storage backend and to begin with recommendation-only operation. The use of Python, Pandas, NumPy, scikit-learn, Streamlit, and SQLite provides a feasible implementation path for a student project.

The Hot, Warm, and Cold categories provide a clear conceptual model for matching files to different storage requirements. However, a model prediction should not automatically override business rules, data-retention obligations, or user-defined exceptions. Any migration feature must preserve data integrity and provide verification and recovery mechanisms.

The effectiveness of the system remains an empirical question. A representative labelled dataset, careful train-test separation, per-class evaluation, and operational testing are required before making claims about accuracy, latency, or cost reduction. The project therefore presents a structured design and evaluation plan that can be extended as implementation results become available.

The project also emphasizes that storage tiering is not only a classification problem. The predicted tier must be interpreted within the context of actual storage capabilities, retrieval requirements, and data-protection policies. For this reason, the proposed design keeps the classifier separate from the migration controller and recommends a review stage before automated movement.

By retaining metadata, prediction timestamps, model versions, and migration history, the prototype can provide a basic audit trail. This information will help the team understand how recommendations are produced and whether a file's classification changes as its access pattern evolves. The same records can support later experiments and model improvements.

In summary, the AI-powered storage system classifier offers a practical way to explore machine learning in storage management. Its success should be judged not only by classification metrics but also by reliability, transparency, safe handling of files, and the usefulness of its recommendations to users. The next step is to implement the pipeline, evaluate it on a clearly documented dataset, and report the observed results.

ACKNOWLEDGMENT

The authors acknowledge the guidance and academic support provided for the Information Storage Management (ISM) project.

REFERENCES

[1]. L. Breiman, “Random Forests,” Machine Learning, vol. 45, pp. 5–32, 2001.

[2]. F. Pedregosa et al., “Scikit-learn: Machine Learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
[3]. pandas development team, “pandas documentation.” [Online]. Available: https://pandas.pydata.org/docs/
[4]. NumPy Developers, “NumPy documentation.” [Online]. Available: https://numpy.org/doc/
[5]. Streamlit, “Streamlit documentation.” [Online]. Available: https://docs.streamlit.io/
[6]. Amazon Web Services, “Amazon S3 storage classes.” [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/storage-class-intro.html
[7]. SQLite Consortium, “SQLite documentation.” [Online]. Available: https://www.sqlite.org/docs.html

 This work is licensed under a Creative Commons Attribution 4.0 International License