2017 IEEE/ACM 39th International Conference on Software Engineering

Can Latent Topics in Source Code Predict Missing Architectural Tactics?

Raghuram Gopalakrishnan*, Palak Sharma*, Mehdi Mirakhorli*, Matthias Galster†
*Rochester Institute of Technology, USA
†University of Canterbury, New Zealand
{rg8772, ps2671, mxmvse}@rit.edu, mgalster@ieee.org

***Abstract*—Architectural tactics such as heartbeat, resource pooling, and scheduling provide solutions to satisfy reliability, security, performance, and other critical characteristics of a software system. Current design practices advocate rigorous up-front analysis of the systems quality concerns to identify tactics and where in the code they should be used. In this paper, we explore a bottom-up approach to recommend architectural tactics based on latent topics discovered in the source code of projects. We present a recommender system developed by building predictor models which capture relationships between topical concepts in source code and the use of specific architectural tactics in that code. Based on an extensive analysis of over 116,000 open source systems, we identify significant correlations between latent topics in source code and the usage of architectural tactics. We use this information to construct a predictor for generating tactic recommendations. Our approach is validated through a series of experiments which demonstrate the ability to generate package-level tactic recommendations. We provide further validation via two large-scale studies of Apache Hive and Hadoop to illustrate that our recommender system predicts tactics that are actually implemented by developers in later releases.**

***Index Terms*—Architectural design and implementation, tactic recommender, emergent design**

I. Introduction

The success of software-intensive systems heavily depends on the extent to which quality concerns such as reliability, availability, security and performance are addressed. Software architects therefore utilize a rich set of proven architectural *patterns* and *tactics* that help satisfy specific quality concerns. Architectural patterns (e.g., layers, pipe-and-filter, blackboard, or publish-subscribe schemes) determine the overall structure and behavior of a software system [16] and are typically selected early during development. While applying a pattern is a strategic decision towards creating a solution to satisfy *multiple* system requirements, architectural tactics are design decisions that improve *individual* and specific quality concerns [6], [8], [27]. Therefore, a pattern usually implements multiple tactics, but tactics can be used independent from patterns [27]. Tactics play an integral role in shaping the high-level design of software since they describe reusable techniques and concrete solutions for satisfying a wide range of quality concerns [8], [25]. For example, reliability tactics provide solutions for fault mitigation, detection, and recovery. Performance tactics provide solutions for resource contention, optimizing response time and throughput. Security tactics provide solutions for authorization, authentication, non-repudiation and other such factors [25]. Tactics are found in the implementation of many performance-centric and safety-critical systems [42]. For instance, the Airbus flight control system uses reliability tactics such as *redundancy with voting*, *heartbeat*, and *checkpointing* to provide solutions for fault mitigation, detection, and recovery. Similarly, the Google Chromium uses performance tactics such as *resource pooling* and *scheduling* to optimize response time and latency [37], [43].

Ideally, relevant architectural tactics are identified in early phases of the design and then implemented alongside functional features. However, in practice this is not always the case. Architectural solutions often evolve during the development process [45]. For example, our analysis of the source code of the *Wavelet Enterprise Management Portal (EMP)* showed that the initial release included a module for editing a customer's personal information including their Social Security Number, credit limit and bank information. However, no *audit trail* (a tactic to achieve secure payment) was implemented until release 6.

Incremental and iterative development and the practice of delivering the simplest functionality first [9] can lead to systems which are, at least initially, not as secure, reliable, or responsive as we would like them to be. In this paper we partially address this problem and explore how to use latent topics in the source code to identify packages in which architectural tactics could be usefully deployed. Most software systems are composed of sets of interacting packages (modules, etc.) which are related to a common goal and common topics. For example, a set of classes in object-oriented source code dealing with online purchases might be described in terms of credit card transactions, user authentication, invoicing, shipping, and inventory management. Our recommender system identifies tactics based on such topics. It uses machine learning and text mining algorithms to mine the source code of 116,000 open source projects and identify the latent source code topics which are correlated with architectural tactics. Then it uses this association to train an inference algorithm and build a recommender system that generates tactic recommendation for software packages

One of the fundamental assumptions of our work is that underlying topics in source code are correlated with the use of architectural tactics. For instance, it is likely that architectural tactics such as *check pointing with roll back* or *authentication* might be found in classes associated with credit card handling.

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.10

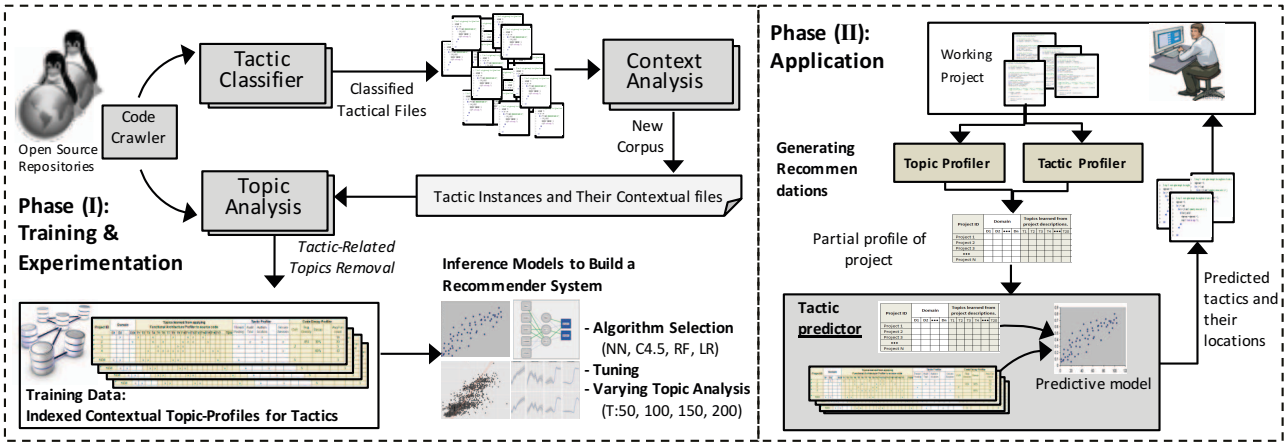


Fig. 1. Overview of the research process.

Similarly, the *heartbeat* tactic might be observed frequently in classes implementing socket-based communication channels. While many such associations are intuitive, to the best of our knowledge there are no previous attempts to analyze them in a systematic way, nor to develop techniques for utilizing this information to recommend areas of code in which tactics should be placed. This approach fits well into the emergent and evolutionary design models that are prevalent today.

In this paper we examine the following two research questions as the first step toward building a tactic recommender system:

> **RQ1:** *Can latent topics in the source code be used to predict the usage of architectural tactics?*

An underlying assumption to study this RQ is that there is a significant amount of collective wisdom embedded in the code of open source systems. While we do not expect each and every system to be implemented in an ideal way, we expect sufficient evidence of proven architectural solutions to extract meaningful relationships between the use of architectural tactics and a wide variety of topics in source code. The results of our study show that in case of nine tactics studied in the paper, there is a strong correlation between latent topics discovered from source code and usage of architectural tactics. This result led to the design of a recommender system. In order to assess the usefulness of that recommender system, we therefore defined a second research question.

> **RQ2:** *Can a tactic recommender system based on latent topics in source code generate useful recommendations?*

To evaluate how "useful" the generated recommendations are, we compare these recommendations with the actual decisions made by developers to implement an architectural tactic. We did this by comparing different versions of two open source software systems where one version did not implement a tactic but a later version fully implemented the tactic. We found that tactics recommended by our approach were indeed implemented in later versions of the open source software systems that we studied.

The **contribution** of our work is two-fold:

- An in-depth analysis of the relationship between *topics* in source code and *tactics* implemented in that code through building inference models.
- Development and evaluation of a novel tactic recommender system trained through analyzing the relationships between topics and tactics from source code of over 116,000 open source system.

While we only investigate nine architectural tactics in this paper, the results of our work highlight the feasibility of building a recommender system based on the relationship between topics in the source code and the use of architectural tactics by developers. The focus on predicting where to place **new** tactics differs from our prior work in which we developed a tactic recommender for identifying **existing** tactic implementations in code [41], [43].

II. Overview of Research Process

The work presented in this paper has two phases (Figure 1):

- Phase (I) – *Training and Experimentation* built a training dataset for the recommender system, conducted experiments to identify appropriate inference algorithms, and evaluated the accuracy of the inference algorithms' predictions of missing tactics in source code. This phase focused on answering research question RQ1.
- Phase (II) – *Application* focused on how the recommender system can be used; experiments were conducted which generated and assessed recommendations for two industrial scale systems. In this phase, we investigated the research question RQ2.

The details of each phase are described in the following.

A. Phase (I): Training and Experimentation

Training and Experimentation requires several components:

1) The *Code Crawler* mines source code from 116,00 open source repositories.
2) The *Tactic Classifier* identifies architectural tactics across 116,000 open source projects and labels the source files implementing these tactics. This classification algorithm has been shown in our prior work to perform well for the task of detecting the tactics independent of projects' underlying programming languages [43]. It outperformed the other algorithms and retrieved most of the tactical code snippets [41].
3) A *Context Analysis* identifies tactic instances and their associated contextual source files (e.g., files interacting with the tactical files) and packages which will be used to learn clues about what source code topics are correlated with a tactic.
4) *Topic Analysis* uses the identified tactic instances and their contextual source files and a map-reduce based version of the Latent Dirichlet Allocation (LDA) [10] algorithm to discover dominant topics associated with each tactic context. From this list of topics, we remove the tactic-related topics so latent domain concepts could emerge. Finally, we model each tactic instance as a combination of the identified topics. This model is called the *topic profile* for the tactic.

The topic profiles established across 116,000 open source projects are used as training data for our machine learning algorithms to predict tactics. In this phase of work, several experiments identified which machine learning algorithms were suitable to build our recommender system. Furthermore, since we used topic modeling, our recommender system is largely dependent on heuristics to determine the parameters for the LDA algorithm. Therefore, we trained and tested the inference algorithms used to build the recommender system with different sets of profile sizes.

Section III describes each of the components required in Phase (I) to obtain the training data. Section IV describes the inference algorithms used to address RQ1, and Section IV-B reports the experiments conducted to compare and tune the inference algorithms and select an appropriate one.

B. Phase (II): Application

After the training data is generated and an inference algorithm is selected and tuned, the recommender system can be used on the source code of new systems.

Ideally, the recommender system will be integrated into a programming IDE. The tactic recommendation process would be initiated when a developer activates the recommender engine and provides the source code of an active project within her/his IDE. This project is automatically parsed and analyzed to create a partial topical profile. Furthermore, the list of existing tactics will be identified so they can be removed from the recommendation list.

In this paper, as a feasibility study, we generate recommendations at the package level, which can be representative of architectural building block. First, a topical profile is generated for each package. Then, these profiles are fed into the trained inference model, which will generate a topic-distribution for each of the given packages and use this model to recommend a tactic.

Section V presents two case studies where the recommender system was applied to different versions of the same system and the generated recommendations were compared with decisions made by the developers in later versions.

III. Creating Topic Profiles for Tactics

A *topic profile* is a matrix that describes the relationship between topics identified in source code and tactics discovered from the open source project.

The profile was constructed by (i) establishing an ultra-large-scale repository of open source projects and downloading their source code (Section III-A), (ii) classifying the source code files into tactical or non-tactical (Section III-B), (iii) identifying tactic instances and their adoption context in each project (Section III-C), (iv) performing topic analysis over tactic instances and their contextual source files to identify topics and pre-processing the topics to remove tactic-related topics so that latent domain concepts could emerge (Section III-D and Section III-E), and finally (v) modeling each tactic instance as a mixture of the identified topics (Section III-F).

Profile ID	Topics learned from applying LDA											Tactics				
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	... Tn	Active Redundancy	Audit Trail	Authen-tication	...	Secure Session
58.373	x			x				x						x		
48.634				x		x				x				x		x
57.635					x				x	x		x				
58.775			x		x	x	x	x	x	x						x
...													x			
57.714		x	x					x	x			x		x		

Fig. 2. Subset of the profiles documenting the topics and associated tactics.

A. Establishing an Ultra-Large Scale Repository

The *Code Crawler* establishes an ultra-large scale repository of software projects extracted from online open source repositories. The current version of our repository contains 116,609 projects extracted from GitHub, Google Code, SourceForge, Apache and Java.net.

Code Crawler utilizes different code crawling applications to retrieve projects from these repositories. To extract the projects from GitHub, we make use of GHTorrent[1] to extract data and events in GitHub in the form of MongoDB data dumps. The dumps include meta-data about projects such as users, comments on commits, programming languages, pull requests, follower-following relations, etc. We also used *Sourcerer* [36], an automated crawling, parsing, and fingerprinting application. *Sourcerer* has been used to extract projects from Google Code, SourceForge, Apache and Java.net. This repository contains versioned source code across multiple releases, documentation (if available), project meta-data, and a coarse-grained structural analyses of each project. We retrieved the entire collection of open source systems from these code repositories. After extracting projects from the above repositories, we performed a data cleaning, removing

[1] http://ghtorrent.org/

redundant, empty or very small projects (i.e., projects that have less than 20 source files). Furthermore, all duplicate forked projects have been identified and removed from this dataset.

B. Tactic Classifier

Each of the downloaded files for a project in the repository was analyzed to determine which, if any, tactics it implemented. As it was infeasible to manually evaluate each file, we used a classification algorithm we had previously developed [41], [43] to detect architectural tactics from source code. These tactics were *audit trail*, *authentication*, *check pointing*, *heartbeat*, load balancing, *role base access control (RBAC)*, *resource pooling*, *scheduling*, and *secure session*. These tactics were selected as they are prevalent across many safety and/or performance-critical systems, and satisfy a range of security, performance, and reliability concerns [3].

The tactic classifier includes three phases: preparation, training, and classifying [43]. In the **preparation phase**, standard information retrieval techniques are used to prepare the data for training and classification purposes. For example, all source code is preprocessed by splitting source code variable names into primitive parts, removing stop words, and stemming words to their root forms. In the **training phase**, a set of textual descriptions of the tactics were processed to produce a set of *weighted indicator terms* that are considered representative of each tactic type. For example, a term such as *priority* is found more commonly in code related to the *scheduling* tactic than in other parts of the code, and therefore receives a higher weight for that tactic. Finally during the **classification phase**, the indicator terms learned in the training phase are used to compute the likelihood that a given source file is associated with a given tactic. Each source file is assigned a score for each tactic, and source files scoring over a pre-defined threshold (t_{cl}) are labeled as being associated with the relevant tactic. In this way, it is possible for a source file to be associated with more than one tactic.

As the tactic classifier had previously been trained to detect the nine tactics used in this study, training was not repeated for this paper. Instead, the source code in our repository was classified using the indicator terms learned from previous training sessions [41], [43]. To compensate for the fact that we could not manually evaluate each tactic, we set high classification thresholds. Based on our previous experiments reported elsewhere [41], [41], the classification threshold t_{cl} of 0.5 obtains a set of tactic-related files with a high degree of confidence.

C. Context Analysis

Implementation of a tactic often spreads across more than one source file. Consequently, domain topics that motivate an architectural tactic are often not fully presented in the tactical file itself but also appear in neighboring files (i.e., files which use tactical files or are used by tactical files). Therefore, we need to identify the whole context in which a tactic is implemented. Latent domain topics are then identified based on that context. The *Context Analysis* is comprised of two steps: First, we identify complete tactic instances whose implementation spreads across multiple tactical files. Second, we consider the full technical context of an implemented tactic instance. The full technical context might include both tactical and non-tactical files:

1) The tactic classifier identifies a list of files which are related to a tactic. However, many of these tactical files may work together to implement a single tactic instance. For instance, two tactical files involved in implementing the heartbeat tactic (one file that monitors a process and another file that is monitored and sends back heartbeats to the monitor) comprise one instance of the heartbeat tactic in the code. Therefore, multiple instances of the heartbeat tactic can be implemented in a project to detect the failure of different processes (i.e, with different source files implementing monitor and sender). To identify tactic instances, we cluster tactical files using Algorithm 1. This algorithm is similar to k-Means clustering [32], but instead of using a similarity score between the clustered items (i.e., tactical files) we use the existence of method calls between the tactical files to move them into the same clusters (i.e., if tactical files call the same method they are put in the same cluster). This creates clusters that contain tactical files which interact with each other.
2) There might be cases in which the technical context can be observed in both a tactical file and the neighboring files which have direct method calls to the tactical file. For example, source files which *use* the authentication function describe the technical context rather than the files which *implement* authentication. Therefore, some latent topics in source code may appear in non-tactical files in the wider context of tactical files. To identify the context, source files classified as tactical or non-tactical are grouped into clusters. Each cluster then defines an instance (i.e., a full implementation) plus additional neighboring files. To discover this full context in which a tactic instance is implemented, we use Algorithm 2 to discover the full context in which each tactic instance is implemented. For each cluster of tactical files, this algorithm identifies the source files that have direct method calls to the tactic instance (i.e., tactical files in a cluster). Then, it groups all the files in the tactic instance (i.e., in the cluster) together with these neighboring files.

The *Context Analysis* component in Figure 1 implements these two algorithms in order to identify the the context for a tactic instance. This component statically analyzes all projects in our repository and generates *new corpus data* that contains the contents of the tactical files themselves and their neighboring files. Based on that corpus data, we identify latent topics in source code during *Topic Analysis* (see Section III-D).

D. Topic Analysis

Our approach utilizes Latent Dirichlet Allocation (LDA) [10] to discover the latent topics from the new corpus data generated in the previous step.

Algorithm 1 Pseudocode to identify tactic instances

procedure TACTICCLUSTERING
  Input: $ClassificationList = \{f_1, f_2, ..., f_n\}$ ▷ *f: Source file*
  Output: $Clusters = \{c_1, c_2, ..., c_k\}$ ▷ *c: cluster of tactical files*
    $L = \{l_f | f = 1, 2, ...n\}$ ▷ *sets of cluster labels for f*
  Initialize $c_i = f_i$, for all $i \in \{1..n\}$
  Initialize $l(f_i) = c_i$, for all $i \in \{1..n\}$
  for each $f_i \in ClassificationList$ **do**
    ▷ *Find which clusters have method calls with a tactical-file & merge them*
    for each $c_j \in Clusters$ **do**
      if $Dependency(f_i, c_j) = TRUE$ **then**
        $MergeList \leftarrow c_j$
    $UpdateClusters(MergeList, Clusters)$
    $Update(l(f_i))$
  return $Clusters$ ▷ *Returns tactic-fragments in form of clusters*

Algorithm 2 Pseudocode to capture context

procedure CONTEXTDATA(CLUSTER T)
  Output: $Context\{c_1, c_2, ..., c_k\}$ ▷ *The new corpus data*
    ▷ *find all neighbors of f*
  $\mathcal{N}BR(f) = \{\forall x \in F : MethodCall(f, x)\}$
  for each $Instance\ t_i \in \mathcal{T}$ **do**
    for each $Source\ File\ f \in t_i$ **do**
      $Context(i) \leftarrow f$ ▷ *Add content of tactical files in tactic instance*
      for each $Source\ File\ s \in \mathcal{N}BR(f)$ **do**
        $Context(i) \leftarrow s$ ▷ *Add content of neighboring files*
  return $Context$ ▷ *To be used by topic analysis*

Topics extracted by the LDA algorithm form two distinct likelihood distribution models. One models the topics discovered from a corpus, indicating how likely a word is to be assigned to a specific topic. The second models each document as a probability distribution indicating how likely it is that a document expresses each of the discovered topics. In our work, given a corpus of documents (the contextual source files from Algorithm 2), the LDA algorithm identifies a set of topics based on word co-occurrences and defines a specific combination of these topics for each tactic instance in our training corpus. These likelihood distributions are used to build the tactic recommender system that predicts and locates tactics in software packages.

E. Tactic-Related Topics Removal

Because the purpose of our work is to **predict** where tactics should be placed and not simply to **detect** where tactics are currently implemented, our approach is designed to focus on topics related to general domain concepts such as credit cards, invoicing, saving files, network communication and even underlying data structures, but not on tactic-related topics. We therefore implemented an additional pre-processing step to remove tactic-related topics from the the list of topics generated in the previous step. Topics generated by LDA are groups of related words. We remove every topic which has terms in common with the tactic *indicator terms* generating by our tactic classifier (Section III-B). The *indicator terms* describe the terms that occur in tactic implementation and have been used to detect tactics in the source code. Removing every topics that has at least one term in common with the tactic *indicator terms*, could lead to the unnecessary removal of topics and therefore negatively impact the training of the inference algorithms. However, using such restrict approach we could obtain a confidence that any residual topics related to actual tactics had been removed. As a result, we were able to create a set of profiles for training purposes which more closely simulated the case in which no tactic had yet been implemented.

F. Construction of Topic Profiles

Once latent topic profiles for each tactic instance were identified, a topic-by-tactic matrix was generated, $M := (M_{i,j})_{P*T}$ where *P* represents the number of topics and *T* the number of identified tactic clusters. A small excerpt of this file is depicted in Figure 2. These profiles were then processed to remove tactic-relate topics, and the resulting profiles simulated a set of "virgin" profiles in which tactics had not yet been implemented. Next, the profiles were converted from a set of topic distribution into a binary distribution. The binary profiles were created by comparing the similarity of each corpus (tactic instance) to the mean of the topic distributions for that corpus.

$$M_{i,j} = \begin{cases} 1 & \text{if } M_{i,j} \geq 1/|T| \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where T is the total number of topics in topic profiles and $M_{i,j}$ is the likelihood of topic j describing tactic instance i.

In next section we use these binary profiles to train a set of inference algorithms and test whether viable tactic recommendation can be generated based on the discovered latent topics from the source code of projects.

IV. BUILDING A RECOMMENDER SYSTEM

Contextual topics for each tactic exhibit various kinds of associations which can be leveraged to develop a tactic recommender system. To determine if topics in source code are effective predictors of tactics, we used the topic profiles to train several machine learning algorithms and evaluated their accuracy in predicting whether a certain tactic should be implemented in the source code of a project. In the following section we first briefly describe these *inference algorithms* and how they can be used in a recommender system. We then present a series of experiments used to select and tune the most appropriate algorithm for a recommender system.

A. Inference Algorithms

From existing machine learning algorithms we chose the Decision Tree (J48), Random Forest, Logistic Regression and Artificial Neural Network which are commonly used in literature to build recommender systems [21], [57], [58].

1) Decision Tree (DT): Decision Trees have been widely used to build recommender systems [22]. A decision tree can form a predictive model which maps a software package to a predicted tactical value based on the topics observed in the package. Each node in the decision tree corresponds to a topic from profiles and each arc from a parent to a child node represents a possible value or a set of values of that topic. This value is driven from the distribution mode generated using LDA.

2) Random Forest (RF): One decision tree only takes advantage of limited information learned from project profiles and topics. Therefore, for many new projects and new topics, a decision tree may produce no predicting result at all. Many researchers have utilized a Random Forest (RF) algorithm to create multiple recommender systems and combine their predictions [58]. Each tree of the forest depends on independent random profiles. Similarly, our approach uses different information from training profiles and topics to construct a random forest to avoid this situation.

3) Logistic Regression (LR): In order to determine if the topics identified by LDA from the source code were effective predictors of missing tactics, we used them as predictors in logistic regression equations to classify whether a certain tactic should be implemented in a software package or not. Logistic regression is a standard classification technique in the experimental sciences. It is also frequently used in software engineering for tasks such as predicting error-prone components in software systems [7], [13].

4) Neural Network (NN): The goal of Neural Network learning is to automatically transform our features (topics) to new sets of complex features that contain more information to distinguish between classes in a training data. We used a 3-layer feed-forward neural network architecture in our work, which consists of 30 artificial neurons in hidden layers, 2 in the output layer and multiple in the input layer based on the number of inputs in training data. These network parameters were selected by trial and error, as used by other researchers [24], [30], [49], [52].

The network is trained using Stochastic Gradient Descent (SGD) with AdaGrad [23], [51], which is effective for both dense and sparse topics (feature) extraction. SGD updates weights associated with each artificial neuron after each training sample until minimum error is achieved. However, the SGD impact is dependent on the use of an appropriate learning rate. To accomplish this, AdaGrad adapts the learning rate as per the frequency of training parameters. For more frequent parameters, the update in learning rate is smaller, empowering the network to extract sparse and non-frequent topics.

B. Algorithm Selection and Tuning

To identify the most appropriate machine learning algorithm, we performed a series of experiments. Furthermore, we investigated the impact of the number of topics on the prediction accuracy of the machine learning algorithms.

TABLE I
NUMBER OF TOPICS REMAINED IN EACH PROFILE

Tactic	Modified Topics
Audit	47, 97, 146, 194
Authenticate	47, 96, 145, 185
CheckPoint	49, 98, 144, 188
HeartBeat	46, 91, 143, 191
LoadBalancing	48, 96, 145, 197
Pooling	50, 100, 143, 192
RBAC	50, 96, 143, 195
Scheduler	48, 95, 146, 194
Session	50, 93, 146, 193

1) Experiment Design: To identify the best performing algorithm, we built four different recommender systems and adopted a standard 10-fold cross validation technique (frequently used to evaluate recommender systems [28]) to predict (or recommend) already known tactics in our dataset. We systematically divided the topic profiles into *n* random folds, with minor adjustments to ensure that each fold did not share any profile from projects in another folds. For each run of the experiment, nine folds were used for training and one for testing. For each project in the test set, a tactic was predicted using the recommender system trained by the profiles in the remaining nine folds. This process was repeated ten times until each fold had been tested, and a complete set of recommendations had been generated. All experiments described in this paper share a common experimental design so that results can be compared across experiments.

For the purpose of training different inference algorithms, we created a balanced dataset of tactical and non-tactical profiles. For each tactical file and its neighboring files we created a design profile describing the latent topics associate with the tactic instance. We also randomly selected the same number of non-tactical files and created the design profiles for them following the process described previously.

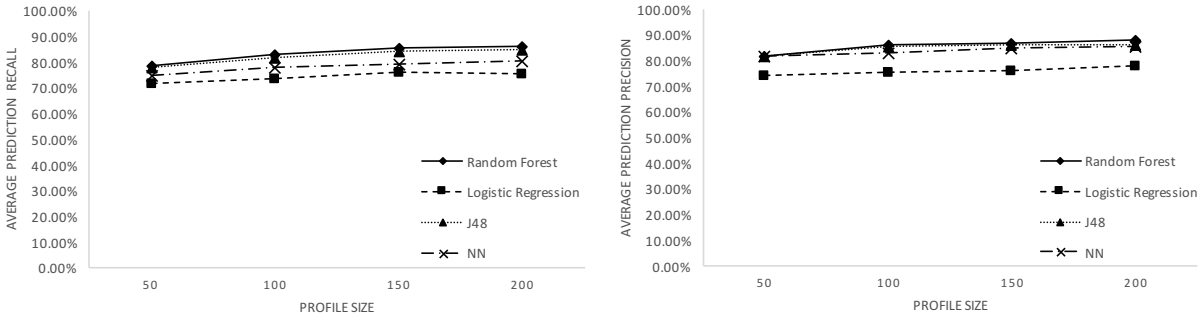
2) Experiment 1: Varying Profile Size: One of the key parameters of the LDA algorithm is the number of topics that should be extracted from the corpus; there is no heuristic to identify the number of topics and needs to be explored based on applications of LDA. The 10-fold cross validation experiment was repeated four times with initial profile sizes of N = 50, 100, 150 and 200. These profiles were generated to identify how many topics should be discovered from each projects to achieve a reasonable accuracy level. As described previously (Section III-E), we removed the tactical topics from each profile so the latent domain topics could emerge. Table I shows the number of topics used in our training profiles after this removal step. Results are reported in the average recall and precision graphs shown in Figure 3. Precision measures the fraction of retrieved files that are relevant and is computed as:

$$Precision = \frac{|RelevantRecs \cap GeneratedRecs|}{|GeneratedRecs|} \quad (2)$$

$$Recall = \frac{|RelevantRecs \cap GeneratedRecs|}{|RelevantRecs|} \quad (3)$$

Precision measures the fraction of the generated tactic recommendations which were correct [20]; Recall measures the fraction of known tactics that were correctly recommended.

Figure 3(a) illustrates average recall of each algorithm for different numbers of topics and across all architectural tactics. Similarly, Figure 3(b) illustrates average precision for each algorithm. While the increase in number of topics resulted in a change in both average precision and recall for each algorithm, this change is not statistically significant.



(a) Average recall across nine tactics.

(b) Average precision across nine tactics.

Fig. 3. Impact of profile size on generating tactic recommendations.

3) Experiment 2: Comparison of Inference Algorithms: The second experiment compared the accuracy of each inference algorithm, to select one to build our tactic recommender system. Because it is not feasible to achieve identical recall values across all runs of the algorithms, the F_1Score computes the harmonic mean of recall and precision and can be used to compare results across experiments:

$$F_1Score = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4)$$

Table II reports the F_1Score obtained by each inference model. We conducted a non-parametric Mann-Whitney U test to compare the performance of these machine learning algorithms. Among all the recommender systems, the predictive model generated using LR performed the weakest. The difference in the obtained F_1Score was significantly less than the other models (p-value=0). The results obtained by RF was higher than the both DC and NN, however this difference was not statistically significant. The gray cells in Table II depicts which inference model performed the best among four.

TABLE II
F_1Score FOR THE COMPARISON OF INFERENCE ALGORITHMS

Tactic	RF	LR	DC	NN
Audit	0.89	0.81	0.88	0.84
Checkpoint	0.90	0.82	0.89	0.86
Authenticate	0.81	0.70	0.80	0.77
HeartBeat	0.92	0.78	0.91	0.89
Load Balancing	0.86	0.77	0.85	0.80
Pooling	0.82	0.71	0.81	0.77
RBAC	0.80	0.70	0.79	0.73
Scheduler	0.86	0.79	0.85	0.82
Session	0.80	0.68	0.79	0.74

C. Analysis of Results

As reported in Figure 3, recall values for the different machine learning algorithms ranged from 0.71 to 0.86, indicating that in general our recommender system was able to recommend a high percentage of known tactics. Precision scores ranged from 0.74 to 0.87 for all algorithms, meaning that the the recommender system was capable of correctly rejecting many non-relevant cases for each tactic.

Experiment on varying profile size suggested that increasing the number of topics from 50 to 100, 150 and 200 will not have significant increase in the accuracy of the inference algorithms. We have only observed minor changes when the number of topics increased from 50 to 100. Considering that RF algorithm performed the best in 10-fold cross validation experiments, we choose the RF algorithm with 100 topics to build a recommender system and apply it to real case studies. In the next section we describe how tactic recommender system can be applied in practice and if it would generate useful recommendations.

V. APPLICATION OF THE RECOMMENDER SYSTEM: TWO CASE STUDIES

To show the practical usefulness of our approach for complex software systems, we used it in two case studies to generate tactic recommendations for the Apache Hadoop and Hive projects, which were not part of the training dataset. Hadoop supports distributed processing of large datasets across computer clusters. It includes over 5000 classes and provides functionality to detect and handle failures to deliver high availability even in the event that underlying clusters fail. Apache Hive, developed by Facebook, has over 4731 classes and is a data warehouse infrastructure built to work with Hadoop providing data summarization, query, and analysis services. We chose these projects because their architectural documentations were available and were available in different versions which allowed us to study the implementation of tactics over time.

A. Tactical Decisions in Hadoop and Hive

In short, we compared the generated recommendations for a version of these systems without a tactic with actual tactical decision made by the developers in subsequent versions. This involved manually identifying two versions of each project:

- $Version_{i,t}$ implements an architectural tactic t
- $Version_{i-n,t}$, a prior version of i where tactic t did not exist; n varies for each tactic

To identify these versions, we took the following steps:

1) We used the tactic classifier to automatically analyze each version of the projects and to classify source files of each version as tactical or non-tactical in order to identify versions that implement a tactic t.

TABLE III
ARCHITECTURE CHANGES IN HIVE: VERSIONS WITH AND WITHOUT TACTICS

Tactic	Package	Rationale
Audit 0.6.0-0.7.0	metastore	audit tactic was added to track user IP and perform metadata operations.
Authenticate 0.6.0 - 0.14.0	ql, service	LDAP is used to authenticate clients.
	jdbc	Uses Simple authentication & Security Layer (SASL) for protecting communication of jdbc driver & server2.
Checkpoint 0.7.0 - 1.0.0	ql	Checkpointing has been added to the job monitor process to log status and locate task failures.
	metastore	Tactic is used to store and fetch aggregated stats
Loadbalancing 0.12.0-0.14. 0	common	Loadbalancing is used to manage multiple HiveServer2 hosts
Pooling 0.3.0 - 0.11.0	metastore, jdbc	Pooling is adopted to enhance performance of the system
	ql,common	Pooling to implement a memory manager for ORC
RBAC 0.6.0 - 1.0.0	ql,common, metastore	Access control was used to determine if the user has access to the processed file.
Scheduling 1.0.0 - 2.0.0	common, ql, service	Scheduling was implemented to improve the performance of queries.
Session 0.13.1-1.2.0	service,jdbc, common, ql	HiveServer generates cookies and validate the client cookie for all the HTTP requests.
Heartbeat 0.12.0-2.1.0	ql,common, metastore	This tactic is used to monitor the client connection to the server.

2) We manually checked the tactical code of each version to ensure we correctly tag each version as tactical or non-tactical with regards to a tactic t.
3) Further confidence was obtained by a detailed review of the project development documentation [1], [2], release notes and issue tracking systems.
4) Tactical files were linked to packages to which they belong (see also Section V-B). This allows us to report information about whether or not a package includes the implementation of a tactic (e.g., if a file in a package implements a tactic, then the package was labeled as implementing that tactic).

Tables III and IV list the tactical and non-tactical versions of Apache Hive and Hadoop, with the components in which the tactic was implemented as well as the rationale for adopting each tactic.

B. Creating New Topic Profile for Studied Systems

Recommendations are generated at the package level using the existing package structure of Hadoop and Hive. Therefore, topic profiles are created at package level. Typically, the packages are a hierarchy of folders and the appropriate level of granularity of packages and folders is project-specific. For Hive, we used the package structure which is used by the original developers as discussed in Hive's architecture [2]. For Hadoop we also used the package structure according to its documentation. In contrast to Hive, Hadoop uses a finer granularity level to describe their code structure [1].[2]

Each package was automatically parsed to group the content of source files within the packages into a single corpus of text for a package. Unimportant terms were removed (i.e., stop words), and words were stemmed to their root forms. This corpus was fed into LDA to infer topics for the packages. Inference was done based on the distribution of topics in the topic profiles as obtained from the training set (see Section III-D).

[2]The data used in this study can be found at design.se.rit.edu/RecSys/.

TABLE IV
ARCHITECTURE CHANGES IN HADOOP: VERSIONS WITH AND WITHOUT TACTICS

Tactic	Package	Rationale
Audit 0.17.0-0.18.0	dfs	Implemented audit logging to allow monitoring and reconstruction of user activities.
Authenticate 0.20.0-0.21.0	hdfs	Checking the data access on DataNodes.
	fs	Incorporated web authentication for the default servlets.
	security	Authentication for client access to HDFS system.
	mapred	Added job-level security to MapReduce.
	ipc	Collected authentication related metrics for RPC.
Checkpoint 0.20.2-0.23.0	hdfs	Added checkpoint to ensure recovery and updates from stand by node.
	fs	This tactic is included as part of file archieving feature.
LoadBalancing 0.20.2-0.23.0	hdfs	High CPU load was resolved by implementing load balancing.
	metric	
Pooling 0.17.0-0.18.0	mapreduce	Connection pooling is implemented to get the client connection status.
	hdfs	
RBAC 0.20.2-0.21.0	hdfs	Applied Role based authentication on data blocks to enforce access control.
	security	
	ipc	Capturing authentication/authorization status metrics.
Scheduling 0.19.0-0.22.0	mapred	Improvised scheduling for improving the time required for assigining tasks by CapacityScheduler.
	ipc	Scheduling implementation to faciliate data node recovery.
Session 0.20.2-0.23.7	hdfs	Sessions are used to manage access to the data files.
	security	Implemented session management for single sign-on for Hadoop web UI and Hadoop RPC
Heartbeat 0.20.0-0.21.0	hdfs	Improvised heartbeat tactic to reduce block report generation time.
	mapred	

C. Comparing Generated Recommendations with Tactics Adopted by Developers

Recommendations were generated using our approach for all the packages in $Version_{i-n,t}$. Recommended tactics were then compared with the design decisions made by the developers in $Version_{i,t}$. To perform this comparison, we matched the packages of $Version_{i,t}$ to the packages of $Version_{i-n,t}$. In many cases a direct match was possible. In the few cases developers where re-organized the packages structure, we manually identified the files from $Version_{i,t}$ that related to $Version_{i-n,t}$. This allowed us to identify which packages in $Version_{i-n,t}$ became tactical in $Version_{i,t}$.

The results of how accurate tactics are predicted are reported in the confusion matrix of Tables V and VI, which provide a visual representation of recommendation results [18], and depicts true and false positives as well as true and false negatives. True positives are the corresponding cell of Yes row and Yes column, indicating the predicted value matched actual value. The matrix shows, for example, in case of Hive, there were 24 packages involved in version 0.6. Out of these 24 packages, 2 packages needed *audit trail* tactics, and in version 0.7 developers added this tactic. Our approach correctly recommended the *audit trail* tactic to one of these packages. Furthermore, our approach recommended audit for two out of 22 non-tactical packages, resulting in a false positive notification to the developers. The last three rows of these two tables report recall, precision and specificity (the

TABLE V
CONFUSION MATRIX SHOWING RESULTS FROM GENERATING RECOMMENDATION FOR DIFFERENT VERSIONS OF APACHE HIVE PROJECT

Predicted / Actual	Audit 0.6-0.7		Authenticate 0.6-0.14.0		Checkpoint 0.6-0.7		HeartBeat 1.0.0-2.0.0		LoadBalancing 0.12.0-0.14.0		Pooling 0.3.0-0.11.0		RBAC 0.6.0-1.0.0		Scheduler 1.0.0-2.0.0		Session 13.1.0-14.0.0	
	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
No	20	2	19	2	22	0	24	2	26	0	15	0	21	0	29	0	23	0
Yes	1	1	1	2	1	1	2	1	0	1	1	3	1	2	3	0	2	2
Recall	50.00%		66.67%		50.00%		33.33%		100.00%		75.00%		66.67%		0.00%		50.00%	
Precision	33.33%		50.00%		100.00%		33.33%		100.00%		100.00%		100.00%		0.00%		100.00%	
Specificity	90.91%		90.48%		100.00%		92.31%		100.00%		100.00%		100.00%		100.00%		100.00%	

Yes: Tactical; No: Non-Tactical

TABLE VI
CONFUSION MATRIX SHOWING RESULTS FROM GENERATING RECOMMENDATION FOR DIFFERENT VERSIONS OF APACHE HADOOP PROJECT

Predicted / Actual	Audit 0.17.0-0.18.0		Authenticate 0.20.2-0.21.0		Checkpoint 0.20.2-V.0.23		HeartBeat 0.20.2-0.21.0		LoadBalancing 0.20.2-V.0.23.0		Pooling 0.17-0.18		RBAC 0.20.2-0.21.0		Scheduler 0.19.0-0.22.0		Session 0.20.2-0.23.7	
	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes
No	22	1	18	11	28	4	31	1	31	1	22	1	31	1	35	1	32	0
Yes	0	1	0	5	0	2	1	1	1	1	1	1	1	1	1	1	1	1
Recall	100.00%		100.00%		100.00%		50.00%		50.00%		50.00%		50.00%		50.00%		50.00%	
Precision	50.00%		31.25%		33.33%		50.00%		50.00%		50.00%		50.00%		50.00%		100.00%	
Specificity	95.65%		62.07%		87.50%		96.88%		96.88%		95.65%		96.88%		97.22%		100.00%	

Yes: Tactical; No: Non-Tactical

fraction of unrelated and unclassified files) achieved in each case study. Specificity is computed as:

$$Specificity = \frac{|NonRelevantRecs|}{|TrueNegatives| + |FalsePositives|} \quad (5)$$

D. Comparison with A Random-Approach

The performance of our tactic recommender system has been compared with the random prediction of tactics for each case study. We use Matthews Correlation Coefficient (MCC) [39] which is used in machine learning as a measure of evaluating prediction quality and reflects the correlation coefficient between the observed and predicted tactic recommendations; MCC returns a value between 1 and +1. A coefficient of +1 represents a perfect tactic recommendation, 0 means that the recommendation is no better than random recommendation and 1 indicates total disagreement between recommendation and observation. Figure 4 reports the MCC value obtained for our two case studies. These values should be compared with zero, which is the MCC value for random recommendations. In Hadoop, all tactics recommended by our system performed better than the random approach. The Mann Whitney U test indicates that the differences are statistically significant (p-value=0). In Hive, recommendations generated for all the tactics except *scheduling* were better than a random approach. Our approach did not recommend the *scheduling* tactic to any of the Hive packages, achieving recall and precision of zero.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP+FP)(FP+FN)(TN+FP)(TN+FN)}} \quad (6)$$

VI. DISCUSSION

A. Answers to Research Questions and Key Findings

In the introduction we outlined our goals of the study:

- **RQ1: Can latent topics in code predict the usage of tactics?** As shown in Section III, we were able to a) identify latent topics in source code, and b) link topics in source code to instances of implemented tactics so that latent topics in source code can be used to predict the usage of architectural tactics.

> **Key finding 1:** *There is a clear association between latent topics in source code and tactics implemented or not implemented in source code.*

- **RQ2: Can a recommender system generate useful recommendations?** As shown in Section V, the recommender system was able to recommend appropriate tactics. The results show the feasibility of developing a tactic recommender system trained using the data collected from a large number of open source systems to recommend a missing tactic for a given project.

> **Key finding 2:** *A tactic recommender system based on latent topics in source code generates recommendations which are similar to actual design decisions made by developers.*

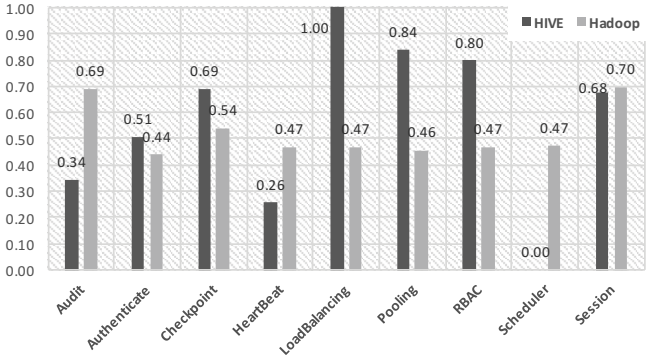


Fig. 4. MCC values for each tactic, comparing generated recommendations with random recommendations.

B. Threats to Validity and Limitations

There are several threats to the validity of this work.

Internal Validity - First, the training set for the recommender system was created using two data-mining techniques, the tactic classifier [41], [43] and LDA, neither of which is 100% precise. Tactic inclusion errors (i.e., false positives) were reduced by setting the threshold value for the tactic detector quite high so that only files detected with high degrees of confidence were selected. LDA was configured manually, however experiments were conducted in order to find the ideal number of topics.

External Validity - A second threat to validity was the construction of inference algorithms and the generalizability of the recommender system. This risk was at least partially mitigated through the use of classic n-fold cross-validation to evaluate the models and demonstrate generalizability. Another limitation of our approach itself is that it only includes nine tactics. However, these nine tactics were selected based on their relevance for building reliable software systems. Our approach could be expanded to cover additional tactics. However, this would require performing additional training based on source code that implements other tactics. The case studies used for evaluation, Hive and Hadoop, both belong to the same domain, which might impact the generalization of the result on different domain datasets.

Construct Validity - Finally, the way tactic-related topics were removed from the topic profiles in order to simulate pre-tactic conditions is another threat. If residual tactic-related topics were left in the profiles, then the inference model might have been trained on the vestiges of the tactics as opposed to the contextual topics found for each tactic. We mitigated this risk by being overly zealous about the removal of tactic related topics and removing any topic that shared a term that our tactic detector had even vaguely associated with a given tactic. Furthermore we validated the efficacy of our recommender system by applying it against the versions of Hive and Hadoop case studies that did not implement tactics. The results were compared with the actual decisions made by the developers in the next releases. Since the examined version did not contain a tactic, our obtained results can provide higher confidence that tactic recommender provides useful recommendations.

VII. Related Work

This paper focused on recommending the use of architectural tactics in projects and packages. Prior work on tactic recommendation has taken a top-down, or requirements-centric view, where architectural tactics [8] or patterns [16] are recommended to address specific quality concerns. Other similar top-down approaches advocate a rigorous analysis of the quality requirements or quality concerns for a system [33] or a more systematic approach to making design decisions [11], [12], [35]. In contrast, our approach is driven by a study of thousands of software projects and focuses on suggesting ways to refactor existing code in order to integrate architectural tactics into an emergent design. In a preliminary work [40] we investigated the relationship between source code topics and tactics in ERP products. The study included over 1000 open source projects and provided empirical evidence that source code topics can predict tactics. However this work was not systematically evaluated. In contrast, this paper uses rigorous empirical design and evaluation to investigate this issue using over 116,000 open source systems. Furthermore, we conduct validations using two large scale industrial systems.

Refactoring existing code to improve the underlying architecture and design [4], [9], [19], [38] can mitigate the problem of architectural erosion [31], [56] or as a normal part of the incremental design process [9]. Numerous authors and practitioners have described refactoring techniques, however these approaches tend to focus on more general tasks such as promoting methods to classes. [47], [53]. Our approach focuses on refactoring for architectural reasons by identifying areas of the code which could be improved through the introduction of proven architectural tactics.

Recommender systems have been studied extensively, originally within the context of e-commerce systems where numerous algorithms have been developed to model user preferences and create predictions. These algorithms vary greatly depending on the type of data they use to create the recommendations. For example, some use content information about the items [46], collaborative data of other users' ratings [50], knowledge rules of the domain [15], or newer highly efficient algorithms such as those based on matrix factorization [34].

In recent years there has been significant emphasis on the use of recommender systems to support a variety of software engineering tasks [26], [48] such as bug triage [5], requirements elicitation [17], source code reuse [55], example recommendation [29], and expertise discovery [44]. However there is little work that combines the use of recommender systems with architectural refactoring. One exception is the work by Terra et al., who proposed a recommender system for identifying architectural violations and then suggesting steps for removing them [54].

Finally, our own prior work has focused on detecting existing tactics [42], [43] however the work in this paper focuses on the more difficult task of predicting packages in which tactics could be usefully employed.

VIII. Conclusions

This paper has presented a first-of-its-kind technique for recommending the placement of architectural tactics within source code packages. In contrast to previous approaches which focus on making architectural decisions to address known quality concerns, our novel approach recommends architectural tactics because a package contains a mix of topics that have previously been associated with the use of that tactic. The results from the inference algorithms and 10-folds cross-validations show that clear associations exist and that these can be used to make useful tactic recommendations.

IX. Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers #CCF-1543176 and #CNS-1629810.

REFERENCES

[1] *Apache-Hadoop Design documents*. http://goo.gl/jmi7Qb.

[2] *Apache-Hive Developer Guide*. http://goo.gl/S3XXqh.

[3] *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012.

[4] P. Abrahamsson, M. A. Babar, and P. Kruchten. Agility and architecture: Can they coexist? *IEEE Software*, 27:16–22, 2010.

[5] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3):10, 2011.

[6] F. Bachmann, L. Bass, and M. Klein. *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*. Technical Report, Software Engineering Institute, 2003.

[7] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Adison Wesley, 2003.

[9] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison Wesley Professional, 2004.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[11] G. Booch. On creating a handbook of software architecture. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 8–8, New York, NY, USA, 2005. ACM.

[12] J. Bosch. Software architecture: The next step. In F. Oquendo, B. Warboys, and R. Morrison, editors, *EWSA*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2004.

[13] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Trans. Software Eng.*, 19(11):1028–1044, 1993.

[14] P. Brusilovsky, A. Kobsa, and W. Nejdl, editors. *The Adaptive Web, Methods and Strategies of Web Personalization*, volume 4321 of *Lecture Notes in Computer Science*. Springer, 2007.

[15] R. D. Burke. Hybrid recommender systems: Survey and experiments. *User Model. User-Adapt. Interact.*, 12(4):331–370, 2002.

[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley and Sons, 2001.

[17] C. Castro-Herrera, C. Duan, J. Cleland-Huang, and B. Mobasher. A recommender system for requirements elicitation in large-scale software projects. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1419–1426. ACM, 2009.

[18] T. Fawcett. Roc graphs: Notes and practical considerations for researchers. *ReCALL*, 31(HPL-2003-4):1–38, 2004.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology International, Inc., 1999.

[20] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

[21] N. Golbandi, Y. Koren, and R. Lempel. Adaptive bootstrapping of recommender systems using decision trees. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 595–604, New York, NY, USA, 2011. ACM.

[22] N. Golbandi, Y. Koren, and R. Lempel. Adaptive bootstrapping of recommender systems using decision trees. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 595–604, New York, NY, USA, 2011. ACM.

[23] A. T. Hadgu, A. Nigam, and E. Diaz-Aviles. Large-scale learning with adagrad on spark. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2828–2830. IEEE, 2015.

[24] N. Haghdadi, A. Zarei-Hanzaki, A. Khalesian, and H. Abedi. Artificial neural network modeling to predict the hot deformation behavior of an a356 aluminum alloy. *Materials & Design*, 49:386–391, 2013.

[25] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Series in Software Design Patterns, 2007.

[26] H. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *RSSE*, pages 11–15, 2008.

[27] N. B. Harrison and P. Avgeriou. How do architecture patterns and tactics interact? a model and annotation. *Journal of Systems and Software*, 83(10):1735 – 1758, 2010.

[28] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1):5–53, Jan. 2004.

[29] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32:952–970, 2006.

[30] D. Hunter, H. Yu, M. S. Pukish III, J. Kolbusz, and B. M. Wilamowski. Selection of proper neural network sizes and architecturesa comparative study. *IEEE Transactions on Industrial Informatics*, 8(2):228–240, 2012.

[31] C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *ESEM*, pages 449–451. IEEE Computer Society, 2007.

[32] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[33] R. Kazman, M. Klein, and P. Clements. Atam: A method for architecture evaluation. *Software Engineering Institute*, 2000.

[34] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[35] P. Kruchten. An ontology of architectural design decisions. In *Groningen Workshop on Software Variability management*, pages 55–62, 2004.

[36] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, 2009. 10.1007/s10618-008-0118-x.

[37] M. Mirakhorli and J. Cleland-Huang. A decision-centric approach for tracing reliability concerns in embedded software systems. In *Proceedings of the Workshop on Embedded Software Reliability (ESR), held at ISSRE10*, November 2010.

[38] J. Madison. Agile architecture interactions. *IEEE Software*, 27:41–48, 2010.

[39] B. W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[40] M. Mirakhorli, J. Carvalho, J. Cleland-Huang, and P. Mder. A domain-centric approach for recommending architectural tactics to satisfy quality concerns. In *Twin Peaks of Requirements and Architecture (TwinPeaks), 2013 3rd International Workshop on the*, pages 1–8, July 2013.

[41] M. Mirakhorli and J. Cleland-Huang. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Transactions on Software Engineering*, 42(3):205–220, 2016.

[42] M. Mirakhorli, P. Maeder, and J. Cleland-Huang. Variability points and design pattern usage in architectural tactics. In *Foundations of Software Engineering (FSE)*, 2012.

[43] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Çinar. A tactic-centric approach for automating traceability of quality concerns. In *ICSE* [3], pages 639–649.

[44] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In W. Tracz, M. Young, and J. Magee, editors, *ICSE*, pages 503–512. ACM, 2002.

[45] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34:115–117, 2001.

[46] M. J. Pazzani and D. Billsus. Content-based recommendation systems. In Brusilovsky et al. [14], pages 325–341.

[47] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proceedings of the International Workshop on Mining Software Repositories*, 2005.

[48] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27:80–86, 2010.

[49] P. Sathiya, K. Panneerselvam, and M. A. Jaleel. Optimization of laser welding process parameters for super austenitic stainless steel using artificial neural networks and genetic algorithm. *Materials & Design*, 36:490–498, 2012.

[50] J. B. Schafer, D. Frankowski, J. L. Herlocker, and S. Sen. Collaborative filtering recommender systems. In Brusilovsky et al. [14], pages 291–324.

[51] A. Senior, G. Heigold, K. Yang, et al. An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6724–6728. IEEE, 2013.

[52] S. R. Shahamiri and S. S. B. Salim. Real-time frequency-based noise-robust automatic speech recognition using multi-nets artificial neural

networks: A multi-views multi-learners approach. *Neurocomputing*, 129:199–207, 2014.

[53] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 175–184, New York, NY, USA, 2007. ACM.

[54] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. *Ninth European Conference on Software Maintenance and Reengineering*, 0:335–340, 2012.

[55] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 204–213. ACM, 2007.

[56] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17:277–306, July 2005.

[57] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[58] H.-R. Zhang and F. Min. Three-way recommender systems based on random forests. *Knowledge-Based Systems*, 91:275 – 286, 2016.