Data Descriptor

The Dataset of the Experimental Evaluation of Software Components for Application Design Selection Directed by the Artificial Bee Colony Algorithm

Alexander Gusev [1], Dmitry Ilin [2] and Evgeny Nikulchev [2,*]

[1] Russian Academy of Education, Data-Center, 119121 Moscow, Russia; alexandrgsv@gmail.com
[2] MIREA—Russian Technological University, Institute of Integrated Safety, Security and Special Instrumentation, 119454 Moscow, Russia; i@dmitryilin.com
* Correspondence: nikulchev@mail.ru

Received: 7 May 2020; Accepted: 6 July 2020; Published: 8 July 2020

Abstract: The paper presents the swarm intelligence approach to the selection of a set of software components based on computational experiments simulating the desired operating conditions of the software system being developed. A mathematical model is constructed, aimed at the effective selection of components from the available alternative options using the artificial bee colony algorithm. The model and process of component selection are introduced and applied to the case of selecting Node.js components for the development of a digital platform. The aim of the development of the platform is to facilitate countrywide simultaneous online psychological surveys in schools in the conditions of unstable internet connection and the large variety of desktop and mobile client devices, running different operating systems and browsers. The module whose development is considered in the paper should provide functionality for the archiving and checksum verification of the survey forms and graphical data. With the swarm intelligence approach proposed in the paper, the effective set of components was identified through a directional search based on fuzzy assessment of the three experimental quality indicators. To simulate the desired operating conditions and to guarantee the reproducibility of the experiments, the virtual infrastructure was configured. The application of swarm intelligence led to reproducible results for component selection after 312 experiments instead of the 1080 experiments needed by the exhaustive search algorithm. The suggested approach can be widely used for the effective selection of software components for distributed systems operating in the given conditions at this stage of their development.

Dataset: DOI: 10.17632/3rh3r2hckr.3.

Dataset License: CC-BY 4.0.

Keywords: swarm intelligence; quality of systems and programs; Node.js; software system development; digital platforms; evolutionary computation; computational experiments

1. Introduction

In previous decades, a significant amount of research was devoted to the development of an optimal modular architecture for component-oriented programming [1] based on a set of criteria [2] including the quality assessment of modular software architecture [3], increasing the productivity of modular software by various methods, including clustering methods [4], genetic algorithms [5], and other evolutionary algorithms [6]. These studies were aimed at the a priori optimization of component-oriented software

architecture in terms of the structural connectivity of the modules, and provided preliminary expert assessments of the functional completeness of the selected components. The modern spread of framework-based architecture requires new approaches to numerical measurements of the quality of service provided by cloud-based software products [7], i.e., the degree to which the product meets the stated and implied needs when used under specified conditions. Thus, the preference for a component should be based on an experimental study of the quality of its operation in a stack together with the components implementing the rest of the functionality of the software system. It is also necessary to guarantee the reproducibility of the experiments and the conformity of the experimental environment to the real operating conditions of the software system, which can be done through using the preconfigured virtual infrastructure, simulating the desired operating conditions [8].

However, two crucial issues arise with organizing experiments to assess the quality of the operation of software components. The first issue is the potential long execution of the experimental algorithm depending on the development goals. The second one is the need to guarantee the sustainability of the solution. Thus, the experimental assessment for each stack of software components should be performed multiple times to ensure that the results can be reproduced and that the impact of random factors such as delays in reading from hard drives, data transfer delays, etc. is minimized. It should be noted that the majority of stacks would in practice show mediocre results in a short sequence of experiments and there would be no need to evaluate them further. Thus, swarm intelligence can be applied to control the process of selection as it becomes possible to make sure that the best results are reproduced while the mediocre stacks are quickly removed from consideration by the swarm.

Thus, the aim of our research was to develop a time-efficient and sustainable swarm intelligence approach to control the process of the selection of software components based on experimental evaluation of their quality of operation and apply it to the practical task of component selection for the Digital Psychological Tools (DigitalPsyTools.ru) for Conducting Large-Scale Psychological Research at the Russian Academy of Education [9]. Among the swarm intelligence algorithms, the artificial bee colony (ABC) algorithm was chosen as it shows good convergence in the tasks of software development optimization, including effort estimation [10], feature selection [11], requirement optimization [12], code coverage [13], and distributed database query optimization [14].

This data article provides the experimental results for the quality evaluation of Node.js 12.16.2 components directed by the artificial bee colony algorithm (ABC) [15]. The experiments were performed in a virtual infrastructure simulating the desired operating conditions of the software system being developed [16]. The Vagrantfile and Ansible playbook of the virtual infrastructure are provided within the dataset and allow other researchers to reproduce the experimental conditions on their system. During the experiment, alternative sets of Node.js components performed the experimental algorithm in the virtual infrastructure and the Node.js "process" object was used to obtain quality indicators during the initialization and execution phases of the experimental algorithm. The set of quality indicators presented in the dataset includes process.memoryUsage(), process.cpuUsage(), process.hrtime() and their nested indicators for both the phases of the evaluation, which amounts to 14 indicators in all. The dataset contains the scenarios for the software component selection process ("comands.zip") and the experimental algorithm ("service-1.zip"), both of which are to be run in the virtual infrastructure. The modified artificial bee colony algorithm with the cost function script is provided as well ("matlab.zip"). The file "snippets.zip" contains a number of files representing the available software component implementations. The file "logging.js" provides the means for tracking resource utilization during the experiment. The dataset also contains the MATLAB fuzzy inference system file ("StackQual.fis") which was used to assess the overall quality of the software components based on the following three indicators: execution.hrtime, execution.cpuUsage.user, and execution.memoryUsage.rss. The experimental results in the dataset are organized in json files, each of which is associated with an ABC agent that initiated the evaluation. The first number in the filename for each set of results represents the iteration number of the algorithm from 0 to 15, the second one is the serial number of the food source in the population, and then the role of the agent during the

food source evaluation is specified. The MATLAB environment is preserved in "environment.mat" to show the terminal solution set, the parameters of the ABC, and the evolution process in detail.

2. Model

At the first stage the n functional requirements $q_i, i \in [1, n]$ of the software system should be identified as well as the t different configurations ω^k, $k \in [1, t]$ of the virtual infrastructure, representing the set of desired operating conditions for the software system. The software developer then identifies the set of M software components available for the research. Each component should implement at least one of the requirements q_i and may be provided by various third-party providers. The subset of alternative software components from M capable of implementing the requirement q_i is denoted as $m_i, i \in [1, n]$. The sets of software components in which for every functional requirement $q_i, i \in [1, n]$ there exists at least one software component from M are defined as stacks s^j, $j \in [1, p]$. S is the set of all the possible stacks. To evaluate the quality of operation for a stack, the f values, denoting experimentally evaluated partial quality indicators $r_\xi^{k,j}$, $\xi \in [1, f]$, are introduced. Their values belong to the space $\mathbb{R}^f$. Thus,

$$\forall \omega^k : s^j \to R^{k,j} \in \mathbb{R}^f$$
$$R^{k,j} = \left(r_1^{k,j}, r_2^{k,j}, \ldots, r_\xi^{k,j}, \ldots, r_f^{k,j}\right)^T, k \in [1, t], j \in [1, p],$$

where $r_\xi^{k,j}, \xi \in [1, f], k \in [1, t], j \in [1, p]$ are the values of experimentally evaluated partial quality indicators for the configuration ω^k of the virtual infrastructure and the stack s^j being evaluated.

The integral quality assessment of the stack is then done by a fuzzy inference system (FIS), to the inputs of which the indicators $r_\xi^{k,j}, \xi \in [1, f], k \in [1, t], j \in [1, p]$ are sent, and the integral quality indicator $\Psi\left(\omega^k, s^j\right)$ for the configuration ω^k of the virtual infrastructure and the stack s^j is produced as the output value of the FIS.

The output value of the FIS is then used by the swarm intelligence algorithm as the cost function for the stack s^j for the configuration ω^k.

The effective selection of software components based on the experimental evaluation of the quality of operation for the chosen configuration of the virtual infrastructure ω^k is aimed at the selection of the stack s^* satisfying the following condition:

$$s^* = \underset{s^j, j \in [1, p]}{\operatorname{argmax}} \Psi\left(\omega^k, s^j\right). \quad (1)$$

3. Method

The artificial bee colony algorithm (ABC) is an optimization method that mimics the behavior of honeybees collecting nectar [15]. The main components of the behavior model of a swarm of honeybees are as follows:

- Food sources: the value of the food source depends on many factors, such as proximity to the hive, nutritional value, and the ease of extracting the nectar.
- Recruited bees: these are agents associated with a particular food source on which they are "employed". Agents transfer information about their source, the distance to it, and its profitability and are likely to share this information.
- Non-recruited bees: agents are constantly on the lookout for new food sources. Two types of non-recruited bees are distinguished: scouts, which explore the environment around the hive in search of new food sources, and onlookers, which wait in the hive and master a new food source using information shared by the recruited bees.

To apply the ABC algorithm to solve Equation (1), the encoding mapping $S \rightarrow \Lambda \subseteq \mathbb{N}^n$ is introduced. Thus, for each stack s^j, $j \in [1, p]$ there are corresponding coordinates for the food source $\mu^j = D(s^j)$, $j \in [1, p]$. Then the reverse mapping $D^{-1} : \Lambda \rightarrow S$ is introduced to transform the coordinates of the food source into the corresponding software stack. The array of stacks being evaluated at the ith iteration of the ABC algorithm is denoted as $\vartheta_\gamma, \gamma \in [1, \Gamma]$; here Γ is the sequence number of the last (terminal) iteration of the ABC algorithm, provided that $|\vartheta_\gamma| \le p, \gamma \in [1, \Gamma]$. The food sources are then denoted as $\mu_\gamma^\eta = \left(\beta_{1_\gamma}^\eta, \ldots, \beta_{n_\gamma}^\eta\right)^T, \eta \in [1, |\vartheta_\gamma|]$; here every $\beta_{i_\gamma}^\eta$ takes its values in the range from 1 to $|m_i|$, which corresponds to the index number of the chosen software component from m_i. The cost function values for $\mu_\gamma^\eta = \left(\beta_{1_\gamma}^\eta, \ldots, \beta_{n_\gamma}^\eta\right)^T, \eta \in [1, |\vartheta_\gamma|], \gamma \in [1, \Gamma]$ coincide with $\Psi\left(\omega^k, s_\gamma^\eta\right)$ being produced by the FIS after the evaluation of the corresponding stack in the experimental environment.

Thus, the task of selecting the stack s^* (1) transforms into the task of selecting the stack s_Γ^* for which the integral indicator Ψ is maximal among the array of stacks $s_\Gamma^\eta, \eta \in [1, |\vartheta_\gamma|]$, corresponding to the set of food sources ϑ_Γ of the terminal iteration of the ABC algorithm:

$$s_\Gamma^* = \underset{s_\Gamma^\eta, \eta \in [1, |\vartheta_\gamma|]}{\operatorname{argmax}} \Psi\left(\omega^k, s_\Gamma^\eta\right) \quad (2)$$

The termination of the ABC algorithm solving Equation (2) is complete when the absolute value of change in the best cost does not exceed the given convergence threshold for the given number of consecutive iterations (maximum number of stall iterations).

After the termination of the ABC algorithm the coordinates of the best food source are transformed into its corresponding software stack using the reverse mapping D^{-1}.

4. The Application of the Approach in Software Development

With the approach introduced above, let us consider the case of selecting Node.js components for the development of the Digital Psychological Tools for Conducting Large-Scale Psychological Research in Russia.

The aim of the platform design is to facilitate countrywide simultaneous online psychological surveys in schools. Due to the unstable internet connectivity in the villages and remote territories, it is crucial to provide guaranteed data delivery even if the communication channel suddenly breaks down. The questionnaire includes the description structure and may include additional resources such as images. All the images and other resources are downloaded in an archive from the server during the survey process.

To meet the goal of facilitating the surveys, the following set of functional requirements and alternative Node.js components was considered: q_1—"sequentially check all the elements of the array for compliance with the condition and return an array consisting of elements for which the check gave the value 'True'", alternative components: "Lodash", "Underscore"; q_2—"apply the specified function to all the elements of an array, thereby returning a new array consisting of the transformed elements", alternative components: "Lodash", "Underscore", JavaScript language tools; q_3—"return the first element of an array", alternative components: "Lodash", "Underscore"; q_4— "generate the full path to the file or directory based on the specified array of path elements", alternative components: "Path"; q_5—"find and replace a substring in the string passed", alternative components: JavaScript language tools; q_6—"perform archiving of the transferred file array and return the generated Zip archive", alternative components: "Adm-zip", "Jszip", "Zipit"; q_7 —"calculate the Message Digest 5 (MD5) [17] hash for the specified dataset", alternative components: "Hasha", "md5", "Ts-md5"; q_8—"read the data from a file", alternative components: "Fs-Extra", "Fs"; q_9—"read the contents of a directory, returning an array of file and subdirectory names in the directory", alternative components: "Fs-extra"; q_{10}—"recursively read the contents of a directory and return an array of file and subdirectory names in the directory", alternative components: "Recursive-readdir".

Thus, $n = 10$, $p = 216$.

The evaluation of the quality of operation was performed with respect to the $f = 3$ partial quality indicators: ${r_1}^{k,j}$—real time spent on the experiment, ns; ${r_2}^{k,j}$—the microprocessor operating time spent in user code during the experiment, ms; ${r_3}^{k,j}$—the increase in the resident set size noted at the end of the experiment (including heap, code segment, and stack), bytes. When conducting the experiment, the partial indicators were normalized with respect to their maximum values in the experiment and took their values in the segment [0; 1].

The choice of these indicators is explained by the need to select the software stack whose resource consumption in terms of resident set size increase and time spent, both by the microprocessor and physically, is minimal, in order to provide a better user experience on various desktop and mobile devices across the country.

The platform development team designed a Mamdani FIS system (see Figure 1) to assess the integral quality $\Psi\left(\omega^k, s^j\right)$ of the stacks using the three partial quality indicators.

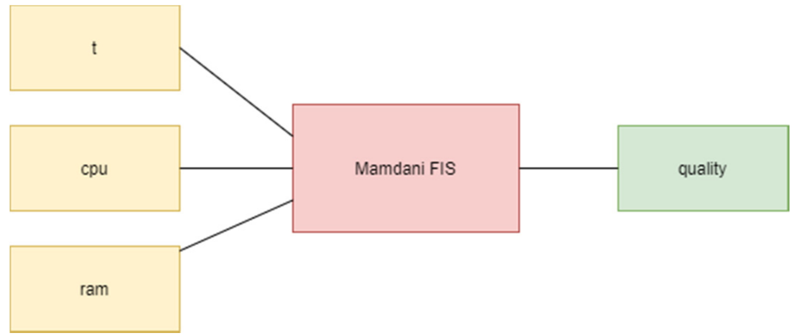
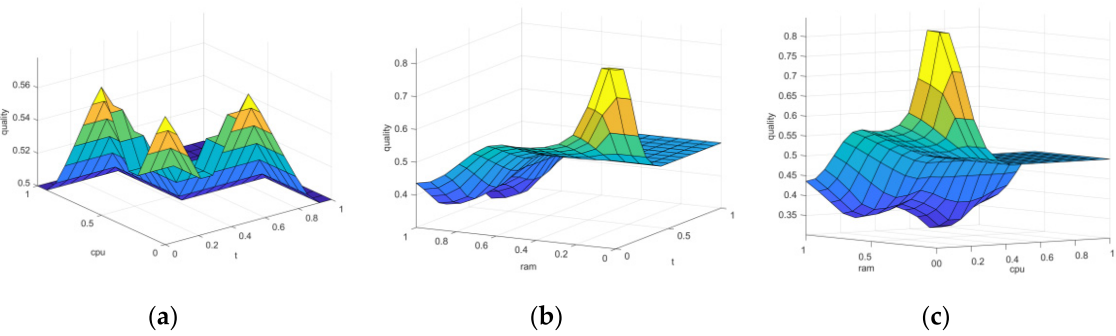


Figure 1. The structure diagram of the fizzy inference system (FIS).

The decision surfaces for the FIS are shown in Figure 2.



(**a**) (**b**) (**c**)

Figure 2. FIS decision surfaces. (**a**) the decision surface with respect to ${r_1}^{k,j}$ and ${r_2}^{k,j}$ (**b**) the decision surface with respect to ${r_1}^{k,j}$ and ${r_3}^{k,j}$ (**c**) the decision surface with respect to ${r_2}^{k,j}$ and ${r_3}^{k,j}$.

The MATLAB Fuzzy Logic Toolbox file of the FIS, "StackQual.fis", is available on the Mendeley repository associated with this paper.

This dataset contains 312 experimental quality evaluations of alternative software stacks with respect to 14 quality indicators as well as configuration files and scenarios to recreate the experimental virtual infrastructure. It also shows in practice the process of the selection of the best software stack by the artificial bee colony algorithm and provides ideas for the further application of swarm intelligence in software component selection.

The dataset can be widely used by software development teams to rapidly build the experimental infrastructure and tune the parameters of the artificial bee colony algorithm according to their task.

The interdependence between the 14 quality indicators can be further investigated to increase the efficiency of the experimental selection of software components.

The experimental results can be used directly in developing software systems with the sets of components which were evaluated.

5. Data Description

The following is the description of the Mendeley dataset, associated with this article:

1. The files "result*.json" provide the individual evaluation of the software stack quality indicators. The first number in the filename represents the iteration number of the ABC from 0 to 15, the second one is the serial number of the food source in the population, and then the role of the agent during the food source evaluation is specified. An example of the inner structure of these files is as follows:


```
{
    "time": 1586854218846,
    "initialization": {
      "cpuUsage": {
        "user": 24000,
        "system": 12000
      },
      "memoryUsage": {
        "rss": 3665920,
        "heapTotal": 3162112,
        "heapUsed": 2988264,
        "external": 23259
      },
      "hrtime": 56721643
    },
    "execution": {
      "cpuUsage": {
        "user": 4000,
        "system": 4000
      },
      "memoryUsage": {
        "rss": 266240,
        "heapTotal": 262144,
        "heapUsed": 311848,
        "external": 56536
      },
      "hrtime": 7281639
    }
}
```


Here, the initialization section represents the quality indicators measured while performing the initialization of the experiment ("init.js" from "commands.zip"), the execution section represents the indicators measured when performing the experimental algorithm ("index.js", "main.js" from "service-1.zip"), and "time" is the timestamp for the results. The items inside the sections mentioned above reflect the measured values of the Node.js "process" object: process.cpuUsage(), process.memoryUsage(), and process.hrtime(), which are described in detail in the Node.js documentation [18].

2. The file "commands.zip" contains scenarios for the initialization of the experimental infrastructure, i.e., virtual machines ("init.js"), preparation of the list of alternative software components ("prepare-alternatives.js"), and launching the experimental algorithm ("run-service-1.js").

3. The file "service-1.zip" contains the scenario of the experimental algorithm ("./src/main.js") and the file "index.js", which is the launch scenario of the experiment and the scenario of data gathering before and after the execution of the experimental algorithm.
4. The file "snippets.zip" contains a number of files representing the available software component implementations. Every file provides the factory function with the component initialization algorithm. The factory function returns a new proxy function, which maps passed arguments to match the component's function signature and vice versa for the output.
5. The file "logging.js" provides the means for tracking resource utilization during the experiment.
6. The file "matlab.zip" contains the integer-valued implementation of the artificial bee colony algorithm ("abc.m") and the cost function computation file "penaltyJS.m" as well as supplementary m-files, which are called automatically to implement the data exchange between the ABC on the host and the virtual experimental environment.
7. "Vagrantfile" and "playbook.yml" are the configuration files for Vagrant and Ansible respectively which can be used to deploy the virtual infrastructure for experiments.
8. "StackQual.fis" is the MATLAB fuzzy inference system (FIS) which was used to assess the overall quality of the software components in our research article based on the following three indicators: execution.hrtime, execution.cpuUsage.user, and execution.memoryUsage.rss.
9. "environment.mat" is the MATLAB environment file storing the terminal population of solutions and the parameters of the ABC.
10. "Evolution.tif" is the graph representing the selection process for software stacks with the ABC and showing the increasing value of overall quality, provided by "StackQual.fis" during the experimental evaluations.

6. Experimental Design, Materials, and Methods

The experimental methodology is presented in Figure 3 [19].

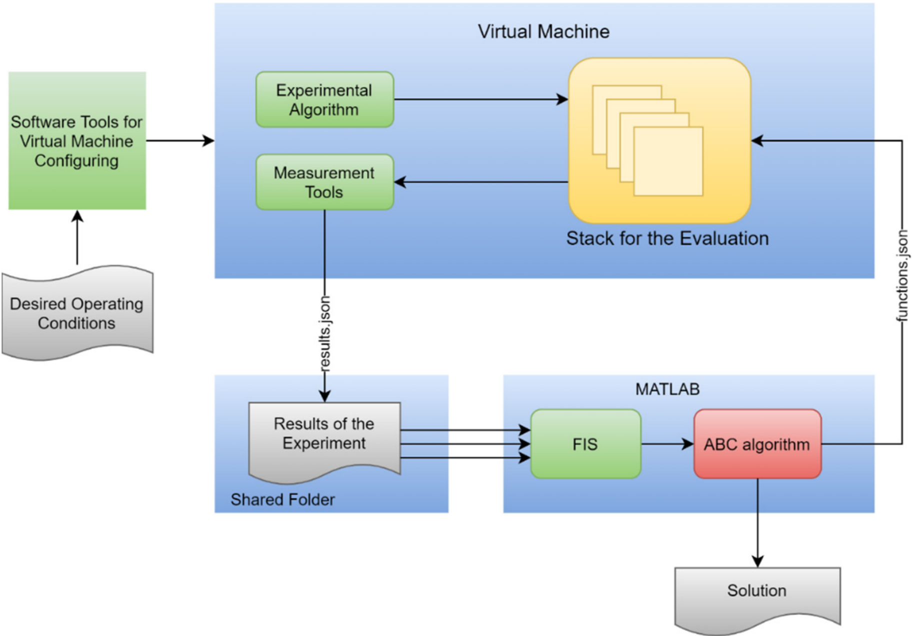


Figure 3. Experimental methodology.

The Virtual Machine was configured with the automatic configuring tools Vagrant and Ansible to simulate the set of desired operating conditions for the software system being developed. The considered configuration included the host central processing unit (CPU): Intel® Core™ i7-7700; number of cores: 4; number of logical processors: 8; clock frequency: 3.60 GHz; the amount of host random access

memory (RAM): 12 GB; the host operating system: Ubuntu 16.04 LTS; Vagrant version: 2.2.4; Node.js version: 12.16.2; 2 virtual CPU cores; 2.0 GB of virtual RAM; virtual machine operating system: Ubuntu 16.04 LTS; provisioning software: Ansible; file exchange tools for the virtual machine: NFS server and BindFS inside the virtual machine; additional system software: git, make, htop, iotop, rsync, and node-gyp.

The above-mentioned parameters of the virtual infrastructure and the Vagrantfile and Ansible playbook file are provided on the Mendeley repository associated with this paper.

The ABC algorithm was executed with the following parameters: colony size: 10; maximum number of stall iterations: 10; convergence threshold: 0.0001; number of onlooker bees: 10; abandonment limit parameter: 20; acceleration coefficient: 1.

At the start of the experiments, the list of alternative sets of software components "alternatives.json" was generated using the "prepare-alternatives.js" scenario from the file "commands.zip", provided with the data repository. The "init.js" scenario was performed to initialize the experimental environment, and "run-service-1.js" was then used to start the experimental infrastructure. The code for the experimental infrastructure and the components are provided within the files "service-1.zip" and "snippets.zip", respectively. At the beginning of each stack evaluation, the ABC algorithm generated the "functions.json" file, which is a json representation of a single stack under evaluation which defines a set of alternative Node.js components in the experiment. The file "functions.json"' contains the code names of the functional features which were to be implemented with the set of components under evaluation, the paths to the components' code, and optionally the parameters of the components. The corresponding MATLAB class "jsfuns.m", which was used to represent the "funstions.json" structure, is available within "matlab.zip" in the data repository associated with this paper.

The integration of the components of the stack was implemented using a functional approach, which is the most convenient way of combining various sets of software components. Each function called during the experiment is a kind of software interface that can be implemented using one of the stack components.

At the initialization stage, the functions defining the basic settings of the components were called. Each of them formed a new anonymous function at the output, which had exclusive access to the component with the specified settings. Next, anonymous functions were placed in a single namespace with the code names of the functional requirements of the software system. To increase the reliability of the results obtained, the component cache (also known as the Node.js module cache) was cleared before initialization.

After initialization, the execution phase of the experimental algorithm began. The experimental algorithm consisted of the following steps:

- Form the path to the directory with a set of subdirectories.
- Read the list of subdirectories.
- Exclude hidden subdirectories.
- Form the path for each directory.
- Do the following for each path:
    - Read the entire list of files recursively.
    - Read and load all the files into the RAM.
    - Create a Zip archive in the RAM.
    - Calculate the MD5 hash for the created archive.

After the execution of the experimental algorithm, a json file, "results.json", was generated. This contained the results of the experiment obtained through the interface of the Node.js "process" object which provided the measurement tools for the evaluation. These data were used as the input values for the FIS. The output value of the FIS was then considered by the ABC algorithm as the cost function for the food source corresponding to the stack that was experimentally evaluated.

7. Results and Discussion

The ABC algorithm converged to the solution of Equation (2) after 15 iterations. The coordinates of the solution were [2 3 1 1 1 1 1 2 1 1], which corresponds to the following selection of components to meet the functional requirements: q_1 should be implemented with the component "Underscore"; q_2 and q_5 should be implemented with the JavaScript language tools; q_3 should be implemented with the component "Lodash"; q_4 should be implemented with the component "Path"; q_6 should be implemented with the component "Adm-zip"; q_7 should be implemented with the component "Hasha"; q_8 should be implemented with the component "Fs"; q_9 should be implemented with the component "Fs-extra"; q_{10} should be implemented with the component "Recursive-readdir". The FIS output for the solution was equal to 0.8123.

The graph of the selection with the ABC algorithm is presented in Figure 4.

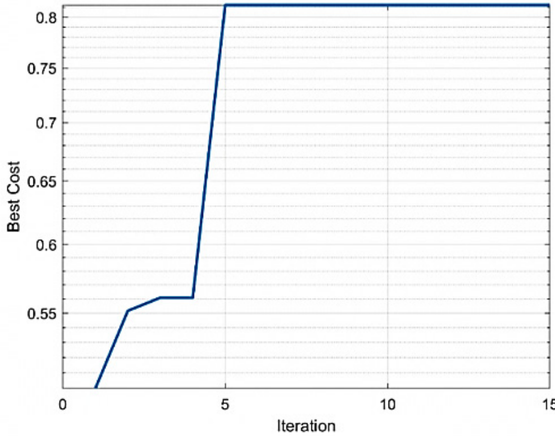


Figure 4. The graph of the selection with the artificial bee colony (ABC) algorithm.

Experimental measurements for the partial quality indicators for all iterations of the ABC algorithm are provided in the Mendeley repository.

The application of the ABC algorithm for selecting the software stack required 312 experimental evaluations of possible stacks before the algorithm converged. The use of extensive search in this task would require 1080 experiments to perform at least five evaluations and compute the average for each stack of 216 considered. The assessment of larger sets of functional requirements and alternative components with extensive search would rapidly lead to a combinatorial explosion (Figure 5).

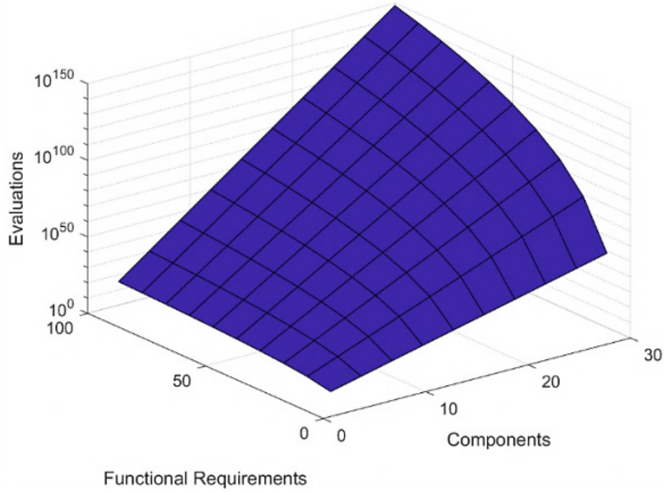


Figure 5. Extensive search combinatorial explosion for larger sets of functional requirements and components. The Z-axis is logarithmic base 10.

Thus, the swarm intelligence approach proposed in this paper allows development teams to effectively select the software stack based on experimental evaluations of reasonable time and acceptable computational efforts, automatically discarding unpromising stacks while verifying, iteration by iteration, the stability of the chosen one.

8. Conclusions

The swarm intelligence approach for the effective selection of software components based on experimental evaluations of their quality of operation was discussed and applied to the problem of the development of a module facilitating psychological surveys for the countrywide Digital Psychological Tools for Conducting Large-Scale Psychological Research. It was shown how software developers can set the desired quality indicators and perform a search for the appropriate set of software components using the virtual infrastructure, simulating the planned operating conditions of the software system. To reduce the number of experiments the developers can use the swarm intelligence approach presented in the paper.

The aim of the development of the Digital Psychological Tools is to facilitate countrywide simultaneous online psychological surveys in schools in the conditions of unstable internet connection and the large variety of desktop and mobile client devices, running different operating systems and browsers. The module whose development is considered in the paper should provide the functionality for the archiving and checksum verification of the survey forms and graphical data. There were 216 possible sets of software components available to build the module. With the experimental approach proposed in the paper, the effective set of components was identified based on evaluations of three quality of operation indicators. To simulate the desired operating conditions and to guarantee the reproducibility of the experiments, the virtual infrastructure was configured, and the artificial bee colony algorithm was applied to reduce the number of experiments with the unpromising sets of software components. The application of the artificial bee colony algorithm led to reproducible results regarding component selection after 312 experiments instead of the 1080 experiments needed by the exhaustive search algorithm.

Author Contributions: Conceptualization, E.N. and A.G.; methodology, D.I. and A.G.; software, D.I.; validation, A.G. and D.I.; investigation, D.I.; resources, D.I.; data curation, A.G. and D.I.; writing—original draft preparation, A.G. and D.I.; writing—review and editing, E.N.; supervision, D.I.; project administration, E.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Russian Foundation for Basic Research (RFBR), grant number 17-29-02198.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Shock, R.C.; Hartrum, T.C. A classification scheme for software modules. *J. Syst. Softw.* **1998**, *42*, 29–44. [CrossRef]
2. Lun, L.; Chi, X.; Xu, H. Coverage criteria for component path-oriented in software architecture. *Eng. Lett.* **2019**, *27*, 40–52.
3. Sarkar, S.; Rama, G.M.; Kak, A.C. API-Based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Softw. Eng.* **2007**, *33*, 14–32. [CrossRef]
4. Mitchell, B.; Traverso, M.; Mancoridis, S. An architecture for distributing the computation of software clustering algorithms. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture WICSA 2001, Amsterdam, The Netherlands, 28–31 August 2001; pp. 181–190. [CrossRef]
5. Kwong, C.K.; Mu, L.F.; Tang, J.F.; Luo, X.G. Optimization of software components selection for component-based software system development. *Comput. Ind. Eng.* **2010**, *58*, 618–624. [CrossRef]
6. Mitchell, B.S.; Mancoridis, S.; Member, S. On the Automatic Modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.* **2006**, *32*, 193–208. [CrossRef]
7. Nazarov, A.N. Processing streams in a monitoring cloud cluster. *Russ. Technol. J.* **2020**, *7*, 56–67. [CrossRef]

8. Kolyasnikov, P.; Nikulchev, E.; Silakov, I.; Ilin, D.; Gusev, A. Experimental evaluation of the virtual environment efficiency for distributed software development. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 309–316. [CrossRef]
9. Kolyasnikov, P.; Nikulchev, E.; Kosenkov, A.; Malykh, A.; Takhirova, Z.; Malykh, S. Analysis of software tools for longitudinal studies in psychology. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 21–33. [CrossRef]
10. Khuat, T.T.; Le, M.H. Applying teaching-learning to artificial bee colony for parameter optimization of software effort estimation model. *J. Eng. Sci. Technol.* **2017**, *12*, 1178–1190.
11. Andaru, W.; Syarif, I.; Barakbah, A.R. Feature selection software development using Artificial Bee Colony on DNA microarray data. In Proceedings of the 2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC), Surabaya, Indonesia, 26–27 September 2017; pp. 6–11. [CrossRef]
12. Alrezaamiri, H.; Ebrahimnejad, A.; Motameni, H. Parallel multi-objective artificial bee colony algorithm for software requirement optimization. *Requir. Eng.* **2020**, 1–18. [CrossRef]
13. Boopathi, M.; Sujatha, R.; Senthil Kumar, C.; Narasimman, S. Quantification of software code coverage using artificial bee colony optimization based on Markov Approach. *Arab. J. Sci. Eng.* **2017**, *42*, 3503–3519. [CrossRef]
14. Panahi, V.; Navimipour, N.J. Join query optimization in the distributed database system using an artificial bee colony algorithm and genetic operators. *Concurr. Comput.* **2019**, *31*, 1–13. [CrossRef]
15. Karaboga, D.; Akay, B. A comparative study of Artificial Bee Colony algorithm. *Appl. Math. Comput.* **2009**, *214*, 108–132. [CrossRef]
16. Basok, B.M.; Zakharov, V.N.; Frenkel, S.L. Iterative approach to increasing quality of programs testing. *Rus. Tech. J.* **2017**, *5*, 12–43.
17. Rivest, R. RFC 1321: The MD5 Message-Digest Algorithm, 1992.
18. Process. Node.js Documentation. Available online: https://nodejs.org/docs/latest-v12.x/api/process.html (accessed on 23 April 2020).
19. Gusev, A.; Ilin, D.; Kolyasnikov, P.; Nikulchev, E. Effective selection of software components based on experimental evaluations of quality of operation. *Eng. Lett.* **2020**, *28*, 420–427.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).