

Singapore Management University

## Institutional Knowledge at Singapore Management University

---

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

---

3-2022

### Efficient search of live-coding screencasts from online videos

Chengran YANG

Ferdian THUNG

David LO

Singapore Management University, davidlo@smu.edu.sg

Follow this and additional works at: [https://ink.library.smu.edu.sg/sis\\_research](https://ink.library.smu.edu.sg/sis_research)



Part of the [Databases and Information Systems Commons](#)

---

#### Citation

YANG, Chengran; THUNG, Ferdian; and LO, David. Efficient search of live-coding screencasts from online videos. (2022). *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, Honolulu, HI, USA, March 15-18*.

Available at: [https://ink.library.smu.edu.sg/sis\\_research/7696](https://ink.library.smu.edu.sg/sis_research/7696)

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email [cherylds@smu.edu.sg](mailto:cherylds@smu.edu.sg).

# Efficient Search of Live-Coding Screencasts from Online Videos

Chengran Yang, Ferdian Thung\*, David Lo  
{cryang, ferdianthung, davidlo}@smu.edu.sg

**Abstract**—Programming videos on the Internet are valuable resources for learning programming skills. To find relevant videos, developers typically search online video platforms (e.g., YouTube) with keywords on topics they wish to learn. Developers often look for live-coding screencasts, in which the videos’ authors perform live coding. Yet, not all programming videos are live-coding screencasts. In this work, we develop a tool named PSFinder to identify live-coding screencasts. PSFinder leverages a classifier to identify whether a video frame contains an IDE window. It uses a sampling strategy to pick a number of frames from an input video, runs the classifier on these frames, and then determines whether the video is a live-coding screencast based on frames classified as containing IDE window. In our preliminary experiment, PSFinder can effectively identify live-coding screencasts as it achieves an F1-score of 0.97.

**Index Terms**—classification, live-coding screencast, search

## I. INTRODUCTION

Many live-coding screencasts are available on the Internet. For instance, the Massive Open Online Courses (MOOC) websites (e.g., Coursera<sup>1</sup>), the nonprofit community that provides coding help (e.g., freeCodeCamp<sup>2</sup>), and millions of programming tutorials in YouTube [1] share their content in the form of live-coding screencasts. In live-coding screencasts, we can see people programming in real-time, which guides them to quickly grasp programming knowledge and get familiar with programming technology. Compared to other programming videos, the live-coding screencast is preferred because it makes programming easy to understand and demonstrates good programming habits [2]. In practice, developers treat the source code in live-coding screencasts as important source of information and reference [3], but developers have difficulty in interacting with the code as they need to transcribe the code first.

While recent work has proposed to automatically extract code from live-coding screencasts to ease developer interaction with code [1], [4], [5], they obtained these live-coding screencasts either by handpicking high quality playlists of programming tutorials [1] or by manually searching for live-coding screencasts [4], [5]. These manual processes are time-consuming. The vast amount and different types of video resources on the Internet makes it difficult to search for live-coding screencasts. As it is hard to accurately guess whether a video is a live-coding screencast from its title and preview image, developers may need to go through many videos

returned by the search result one-by-one until they find live-coding screencasts.

Considering the above scenario, we develop PSFinder to automatically identify live-coding screencasts from a set of videos. For this identification, PSFinder takes a number of video frame samples from the video. It then uses a classifier to determine if a video frame contains an Integrated Development Environment (IDE) window. Based on the frames showing IDE windows, the PSFinder identifies the video as a live-coding screencast. The identified live-coding screencasts can then be passed to any existing video-to-code extraction tools [1], [5] to automatically provide the source code to developers. Therefore, PSFinder complements these video-to-code extraction tools.

As a preliminary experiment, we collect a dataset of 112 videos sampled from the top-5 videos returned by YouTube when querying using the top-100 Java library names in Maven Repository<sup>3</sup>. Specifically, 80 live-coding screencasts covering different kinds of IDEs and 32 non live-coding screencasts are included in this dataset. Using stratified random sampling, we take 80% of the videos as training data and 20% of them as test data. PSFinder achieves an F1-score of 0.97, which is better than a random baseline or a baseline that identifies all videos as live-coding screencasts. Our analysis also found that PSFinder is capable to identify live-coding screencasts against non live-coding screencasts which frames contain IDE screenshots.

Considering that online video platforms generate a huge amount of video every day (eg., more than 500 hours of contents are uploaded to YouTube every minute<sup>4</sup>), PSFinder can be used by a video analysis solution to more efficiently find and process live-coding screencasts from enormous amount of fast-growing online video resources. The larger the dataset size is, the more time PSFinder can save. Additionally, PSFinder can potentially be used as a browser plug-in to optimize and ease the user’s experience when searching live-coding screencasts by providing only videos that are live-coding screencasts in the search result.

The contributions of this paper are:

- We are the first to develop a tool to identify whether a video is a live-coding screencast. It eases the search of live-coding screencasts and complements existing video-to-code extraction approaches [1], [5].

\* Corresponding author.

<sup>1</sup><https://www.coursera.org/>

<sup>2</sup><https://www.freecodecamp.org/>

<sup>3</sup><https://mvnrepository.com/>

<sup>4</sup><https://blog.youtube/press/>

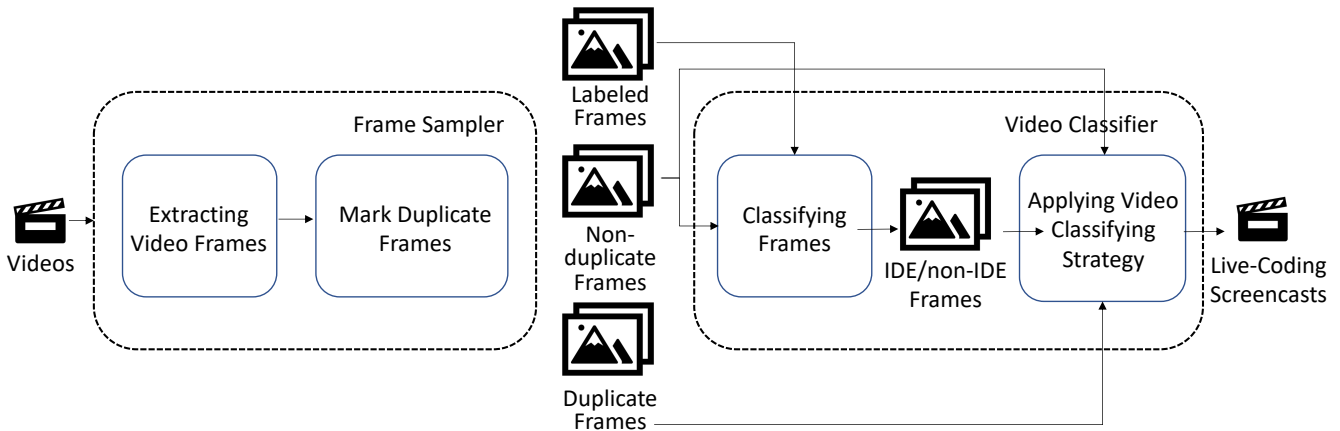


Fig. 1: Overview of PSFinder

- On a dataset of 112 videos with 80 live-coding screencasts and 32 non live-coding screencasts, PSFinder achieves an F1-score of 96.97%, which is better than a random baseline and a baseline that predicts all videos as live-coding screencasts.
- Compared to the baseline, PSFinder can shave off 32% of running time without missing any live-coding screencasts.

The remainder of this paper is structured as follows. We describe PSFinder in Section II. We then describe our experimental settings and research questions in Section III. We present answers to the research questions in Section IV. We present related work in Section V. Finally, we conclude the paper and mention some future work in Section VI.

## II. APPROACH

As shown in Figure 1, PSFinder accepts as input a set of videos that possibly contain live-coding screencasts and outputs the set of videos classified as live-coding screencasts. Firstly, we extract sets of frames from videos and mark duplicate frames using *Frame Sampler*. Duplicate frames are frames that have little or no difference with their preceding frames. After marking the duplicate frames, we feed the non-duplicated frames into *Video Classifier*, which classifies whether the frames contain an IDE window. We call a frame containing an IDE window as an IDE frame. Given the set of IDE frames from a video, *Video Classifier* applies a classification strategy to predict whether the video is a live-coding screencast. Finally, we return the set of videos predicted as live-coding screencasts. We describe *Frame Sampler* in Section II-A and *Video Classifier* in Section II-B.

### A. Frame Sampler

In most cases, analyzing all video frames is not necessary since the contents of consecutive frames are likely to be very similar or identical. Therefore, to reduce processing time, we sample the frames with a rate of one frame per 30 seconds. Then, we mark the duplicate frames. To do so, we utilize the normalized root-mean-square error (*NRMSE*) to compute the dissimilarity of two consecutive frames on the pixel level.

*NRMSE* score ranges from 0 to 1. The score 0 means that the two frames are identical while the score 1 means that the two frames are completely different. Given two frames  $f_i$  and  $f_j$ , *NRMSE* can be computed by following formula:

$$\sqrt{\frac{\sum_{n=0}^{N-1} \sum_{m=0}^{M-1} [f_i(m, n) - f_j(m, n)]^2}{\sum_{n=0}^{N-1} \sum_{m=0}^{M-1} [f_i(m, n)]^2}} \quad (1)$$

where  $f_i(m, n)$  and  $f_j(m, n)$  are the pixel values at row  $m$  and column  $n$  for frame  $f_i$  and  $f_j$ , respectively.  $M$  and  $N$  are the width and the height of the frames. We represent a frame sequence as  $\{f_i\}$  where  $0 \leq i \leq K$ , and  $K$  denotes the last minute of video. For each subsequent frame  $f_j$  ( $j \geq i \geq 1$ ) starting from  $f_i$ , we calculate the dissimilarity of  $f_i$  and  $f_j$  by *NRMSE* and delete successive frame  $f_j$  until the dissimilarity between  $f_j$  and  $f_i$  is above a threshold. We set the threshold to 0.05 rather than 0 to account for small differences that are expected in a live-coding session (e.g, consecutive frames change due to a mouse cursor movement or different time shown in the desktop clock). Finally, non-duplicate frames are fed to the *Video Classifier*.

### B. Video Classifier

After getting non-duplicate frames from *Frame Sampler*, we apply both frame-level classifier and video-level classification strategy to classify videos into two categories: live-coding screencasts and non live-coding screencasts.

1) *Frame-Level Classifier*: We apply a deep learning model to build a frame classifier that categorizes a frame as IDE frame or non-IDE frame. IDE frame is the frame that contains an IDE window while non-IDE frame is the frame that does not contain an IDE window. We use the ViT (Vision Transformer) model [6] composed of transformer encoder architecture to extract image features. ViT model is pre-trained on the ImageNet-21K dataset and has shown excellent performances on image classification tasks. We fine tune ViT model using our labelled frames. By using the fine-tuned

model, each non-duplicate frame is classified to either an IDE or non-IDE frame.

2) *Video-Level Classification Strategy*: Intuitively, live-coding screencasts should continuously show IDE windows over a period of time where developers are interacting with the IDE (e.g., writing and editing the code, debugging the code, etc.). Within this period of time, the IDE frames should change their content, as opposed to displaying the same content (which may indicate that the IDE window is only a screenshot). Thus, PSFinder determine a video as a live-coding screencast by following a two-stage classification strategy: (1) PSFinder detect if there is at least one contiguous sequence of sampled frames with a minimum size of  $s$  that are all non-duplicate frames and classified as IDE frames by ViT models. The size  $s$  signifies the amount of changes we should observe; (2) PSFinder detect whether the proportion of IDE frames to all non-duplicate frames exceeds a threshold  $t$ . A sequence of non-duplicate frames that are all classified as IDE frames signifies that the content of those frames is about IDE and changes over time. If the above two conditions are fulfilled, PSFinder predicts the video as a live-coding screencast.

Given sampled frames  $v$  that are non-duplicate and identified as IDE frames by *Frame-Level Classifier*, PSFinder searches for the shortest contiguous frame sequence  $v_1, \dots, v_j$  where  $s \leq j \leq L$ , and  $L$  denotes the number of sampled frames. By default, we set  $s$  to 4. Also, considering that the number of IDE frames in the sampled frames is  $N_{ide}$  and the number of non-duplicate frames in the sampled frames is  $N_{info}$ , PSFinder will compare the value of  $N_{ide}/N_{info}$  with threshold  $t$ . We set  $t$  to 0.5 since we want at least half of a video to contain IDE frames for it to be classified as a live-coding screencast. For each video, if  $N_{ide}/N_{info} \geq t$  and there exists an consecutive frame sequence,  $v_1, \dots, v_j$ , in which each frame is non-duplicate and identified as IDE frame, PSFinder predict the corresponding video as a live-coding screencast.

### III. PRELIMINARY EXPERIMENT

#### A. Dataset

We have 112 videos in our dataset, including 80 live-coding screencasts and 32 non live-coding screencasts. We obtain 50 live-coding screencasts from *psc2code* [1] dataset that contain the videos with Eclipse IDE and the other 30 live-coding screencasts with other IDEs by searching on YouTube. We also obtain the 32 non live-coding screencasts from YouTube.

We download videos from YouTube that cover live-coding screencasts of other Java IDEs and non live-coding screencasts. To find these videos, we consider a scenario where developers search for live-coding screencasts to learn how to use libraries. Specifically, we consider the top-100 libraries in Maven Repository. For each library, we search YouTube<sup>5</sup> with a query "java <library name>". Considering a real-life scenario where a user is unlikely to go through all the search results from YouTube, we pick only the top-5 videos from

<sup>5</sup><https://www.youtube.com/>

TABLE I: Statistics of the Videos

	# Screencast	# Non-screencast	Total
<b>Train</b>	64	25	89
<b>Test</b>	16	7	23
<b>Total</b>	80	32	112

each search result. From the collection of these videos, we randomly pick 100 candidates for our dataset and download them using Pytube<sup>6</sup>. We manually check whether they are live-coding screencasts, non live-coding screencasts, or videos unrelated to programming (e.g., some video blogs). We find 30 live-coding screencasts, 32 non live-coding screencasts, and 38 videos that are not related to programming. The live-coding screencasts cover mainstream IDEs in the JAVA community (15 covers IntelliJ IDEA<sup>7</sup>, 10 covers Visual Studio Code<sup>8</sup>, and 5 covers Apache NetBeans<sup>9</sup>).

To construct our labeled frame dataset, we extract one frame per second from our video dataset. We use the NRMSE algorithm described in Section II-A to delete identical or highly similar frames. Since the videos vary in length from several minutes to several hours, frames from the long videos would take a large proportion of training data if we do not limit the maximum number of frames extracted per video. Hence, to avoid a possible bias, we set the maximum number of extracted frames from one video to 600. If one video has more than 600 extracted frames, we randomly select the 600 frames.

Next, we label these frames to use for training our frame classifier. Frames coming from non live-coding screencasts are labeled as non-IDE frame since the non live-coding screencasts do not contain IDE. Frames of live-coding screencasts originating from *psc2code* are labeled with IDE-related categories: frames containing IDE window with a clear code area and frames containing an IDE window without clear code area. We relabel these frames as IDE frames as they both contain IDE window. The other frames from *psc2code* dataset are labeled as non-IDE frames. Next, we manually label the frames of live-coding screencasts that are not originating from *psc2code* dataset. These frames are labeled by the first author. Note that labelling IDE/non-IDE frames is an objective task that can be done by a human (with no visual impairment) with perfect (or at least almost perfect) accuracy. It is clear to the labeller whether a frame contains an IDE or not. Therefore, labelling by one person is sufficient. The details of the labeled frames are shown in Table II.

TABLE II: Statistics of the Labeled Frames

	# IDE	# Non-IDE	Total
<b>Train</b>	11,741	11,945	23,686
<b>Test</b>	2,938	1,511	4,449
<b>Total</b>	14,679	13,456	28,135

<sup>6</sup><https://pytube.io/en/latest/>

<sup>7</sup><https://www.jetbrains.com/idea/>

<sup>8</sup><https://code.visualstudio.com/>

<sup>9</sup><https://netbeans.apache.org/>

## B. Experimental Settings

We apply stratified random sampling, drawing 20% videos from live-coding screencasts and 20% from non live-coding screencasts, respectively. These videos are considered the testing data and remaining live-coding screencasts and non live-coding screencasts in our dataset are the training data. The detailed split of our video dataset is shown in Table I. We use FFMPEG<sup>10</sup> to extract frames from the videos. Every frame is converted into a frame of size (300×300) pixels. We use PyTorch to implement our deep learning model. We set the maximum number of training iterations as 100. We use Adam [7] to optimize the model. We run our experiment on a machine with an Intel Core i7 CPU, 64GB memory, and two NVidia RTX 3070 GPU with 8GB RAM.

## C. Research Questions

*RQ1. Can PSFinder effectively identify if a video is a live-coding screencast?*

We run PSFinder and compare it with two baselines. The first baseline randomly classifies a video into live-coding or non live-coding screencasts with 50% chance, which we call as the *random baseline*. The second baseline classifies all videos as live-coding screencasts, which we call as the *all positive baseline*. We evaluate them using Precision, Recall, and F1 score as defined below. We repeat the random baseline twenty times and report the average scores for each metric.

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F1\text{-score} = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

TP occurs when a live-coding screencast is correctly classified as a live-coding screencast. FP occurs when a non live-coding screencast is incorrectly classified as a live-coding screencast. FN occurs when a live-coding screencast is incorrectly classified as a non live-coding screencast.

*RQ2: What are the cases where PSFinder fail to correctly identify live-coding screencasts or non live-coding screencasts?*

We manually look at the misclassified videos. We extract the sampled frames from these videos to assist our analysis. We will consider two aspects: the content of informative frames in the misclassified video; and the similarity of the misclassified video with both live-coding and non live-coding screencasts.

## IV. RESULTS

### A. Effectiveness in Video Classification

Table III shows the performance of PSFinder on identifying live-coding screencasts. Compared with the random baseline, PSFinder improves recall by 92.31%<sup>11</sup>, precision by 42.42%,

TABLE III: Effectiveness of PSFinder

	Recall	Precision	F1 Score
Random Baseline	0.52	0.66	0.58
All Positive Baseline	1	0.73	0.84
PSFinder	1	0.94	0.97

and F1 score by 67.24%. Compared with the all positive baseline, PSFinder has the same recall and improves precision by 28.77%, and F1 score by 11.90%. It indicates that PSFinder effectively identify the live-coding screencasts. More specifically, the recall of PSFinder is 1, which illustrates the reliability of PSFinder in capturing live-coding screencasts.

### B. Analysis of Misclassified Videos

PSFinder incorrectly identifies a non live-coding screencast that contains only IDE screenshots<sup>12</sup> as a live-coding screencast. This video is the second video returned by YouTube when searching using the keywords “java” and “Gpars”. It introduces various use cases of GPars library<sup>13</sup> in Java. The developer used screenshots rather than screen recordings to demonstrate the coding process.

Considering that the other non live-coding screencasts containing IDE screenshots in our test dataset are correctly identified, we analyse how this video differs from the others. We extracted the sampled frames from this video and found that this video shows not only the code screenshots but also a live shot of the presenter in the bottom left corner of the screen. The IDE screenshots in these frames lead to a misclassification by our *frame-level classifier*. The continuous movement of the presenter’s live shot causes these frames not to be considered duplicate frames despite the fact that they show the same IDE screenshot. It appears that our *video-level classification strategy* is unable to handle a large moving objects in the frames (e.g., live shots of the user). In the future, rather than measuring the pixel-level similarity between consecutive frames, we can potentially use OCR (optical character recognition) technologies to extract the code snippet from frames and calculate the similarity between the code snippets to avoid this kind of misclassification.

## V. RELATED WORK

Previous work discussed the role of social media in software development [8], which mentions the increasing role of video podcasts in software development, especially the practical application of video podcasts in teaching programming. Following [8], researchers began to focus on how developers can share and document knowledge by using live-coding screencasts [3]. By analyzing programming videos and interviewing developers, the researchers demonstrate how well live-coding screencasts can transfer knowledge between developers and build reputations for developers who created the videos.

With the rapid development of live-coding screencasts, several kinds of video-to-code tools have been proposed [1],

<sup>10</sup><https://www.ffmpeg.org/>

<sup>11</sup>improvement computed by:  $(\frac{\text{recall of PSFinder}}{\text{recall of random baseline}} - 1) \times 100\%$

<sup>12</sup><https://www.youtube.com/watch?v=angDXZBp1zc>

<sup>13</sup><http://www.gpars.org/>

[9]–[12]. Video-to-code tools leverage shape detection algorithms to find code areas and optical character recognition (OCR) technology to transcribe the code in an image to text. ACE [9] proposes a video-to-code tool which could correct OCR errors by applying statistical language models. Besides, CodeTube [10] utilizes OCR to extract code from frames of live-coding screencasts and computer vision techniques, including image segmentation and shape detection, to recognize the code area in the frame. To provide better experience, Codemotion [11] creates an additional UI window to show the extracted code. After that, *psc2code* [1] points out and eliminates noise issues that was not handled by CodeTube [10] and achieves the state-of-the-art performance.

In addition to extracting code from live-coding screencasts, recent works are also looking at other ways videos can help developers [13]–[15]. V2S [13] is a lightweight tool to obtain and capture useful video information, including bugs or feature requests from screen recordings of mobile applications. Moreover, it can translate video recordings into replayable scenarios. TANGO [14] detects duplicate video-based bug reports. TANGO leverages both visual and textual information to detect videos that are reporting the same bug. To improve the learning efficiency of mobile app development, UIScreens [15] could localize and extract the most representative UI screens in a mobile programming screencast, thus developers would quickly comprehend what an app displayed in a video is about. To the best of our knowledge, while there have been some work on how videos can benefit the developer community, none of them automatically identifies live-coding screencasts.

## VI. CONCLUSION AND FUTURE WORK

We propose PSFinder to identify live-coding screencasts from online videos. From input videos, we sample a set of frames for every video and delete similar frames. Then, we feed the sets of frames into our classifier and determine whether the videos are live-coding screencasts. The evaluation shows that PSFinder can identify live-coding screencast with an F1 score of 0.97. In the future, we plan to evaluate PSFinder using more videos. We also plan to improve our classifications strategy by adding more rules such as learning the minimum number of IDE frames for live-coding screencasts from training videos. Moreover, we plan to utilize the information extracted from videos by PSFinder to help automating software engineering tasks, e.g., library recommendation [16], automated debugging and repair [17], [18], etc.

**Replication Package.** The source code for PSFinder is available at <https://github.com/soarsmu/PSFinder>.

## ACKNOWLEDGMENT

This research/project is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-1-193). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

## REFERENCES

- [1] L. Bao, Z. Xing, X. Xia, D. Lo, M. Wu, and X. Yang, “psc2code: Denoising code extraction from programming screencasts,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–38, 2020.
- [2] A. G. S. Raj, J. M. Patel, R. Halverson, and E. R. Halverson, “Role of live-coding in learning introductory programming,” in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 2018, pp. 1–8.
- [3] L. MacLeod, M.-A. Storey, and A. Bergen, “Code, camera, action: How software developers document and share program knowledge using youtube,” in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 104–114.
- [4] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead, “A deep learning approach to identifying source code in images and video,” in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 376–386.
- [5] M. Alahmadi, A. Khormi, B. Parajuli, H. Hassel, S. Haiduc, and P. Kumar, “Code localization in programming screencasts,” *Empirical Software Engineering*, vol. 25, no. 2, pp. 1536–1572, 2020.
- [6] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.
- [7] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>
- [8] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky, “The (r) evolution of social media in software engineering,” in *Future of Software Engineering Proceedings*, 2014, pp. 100–116.
- [9] S. Yadid and E. Yahav, “Extracting code from programming tutorial videos,” in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016, pp. 98–111.
- [10] L. Ponzanelli, G. Bavota, A. Mocchi, M. Di Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza, “Too long; didn’t watch! extracting relevant fragments from software development video tutorials,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 261–272.
- [11] K. Khandwala and P. J. Guo, “Codemotion: expanding the design space of learner interactions with computer programming tutorial videos,” in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, 2018, pp. 1–10.
- [12] L. Bao, J. Li, Z. Xing, X. Wang, and B. Zhou, “Reverse engineering time-series interaction data from screen-captured videos,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 399–408.
- [13] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, “Translating video recordings of mobile app usages into replayable scenarios,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 309–321.
- [14] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, “It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 957–969.
- [15] M. Alahmadi, A. Khormi, and S. Haiduc, “Ui screens identification and extraction from mobile programming screencasts,” in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 319–330.
- [16] F. Thung, D. Lo, and J. Lawall, “Automated library recommendation,” in *2013 20th Working conference on reverse engineering (WCRE)*. IEEE, 2013, pp. 182–191.
- [17] S. Wang and D. Lo, “Version history, similar report, and structure: Putting them together for improved bug localization,” in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 53–63.
- [18] X. B. D. Le, D. Lo, and C. Le Goues, “History driven program repair,” in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.