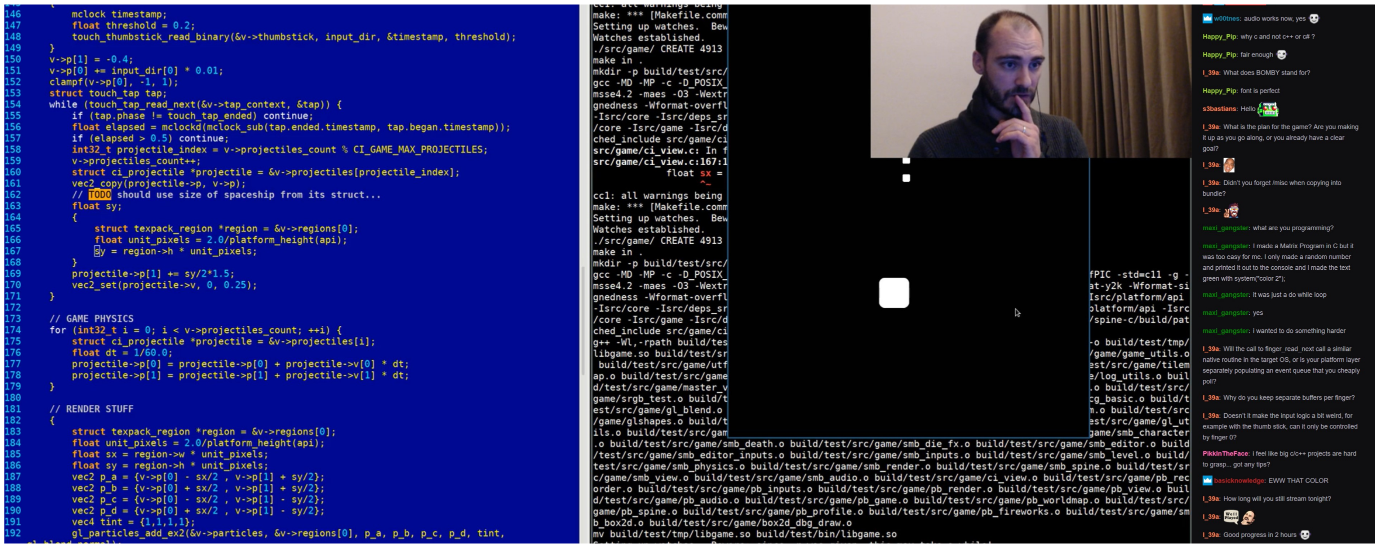
An Exploratory Study of Live-Streamed Programming

Abdulaziz Alaboudi
George Mason University
Fairfax, Virginia, USA
aalaboud@gmu.edu

Thomas D. LaToza
George Mason University
Fairfax, Virginia, USA
tlatoza@gmu.edu



```
        mclock timestamp;
        float threshold = 0.2;
        touch_thumbstick_read_binary(&v->thumbstick, input_dir, &timestamp, threshold);
    }
    v->p[1] = -0.4;
    v->p[0] += input_dir[0] * 0.01;
    clampf(v->p[0], -1, 1);
    struct touch_tap tap;
    while (touch_tap_read_next(&v->tap_context, &tap)) {
        if (tap.phase != touch_tap_ended) continue;
        float elapsed = mclockd(mclock_sub(tap.ended.timestamp, tap.began.timestamp));
        if (elapsed > 0.5) continue;
        int32_t projectile_index = v->projectiles_count % CI_GAME_MAX_PROJECTILES;
        v->projectiles_count++;
        struct ci_projectile *projectile = &v->projectiles[projectile_index];
        vec2_copy(projectile->p, v->p);
        // TODO should use size of spaceship from its struct...
        float sy;
        {
            struct texpack_region *region = &v->regions[0];
            float unit_pixels = 2.0/platform_height(api);
            sy = region->h * unit_pixels;
        }
        projectile->p[1] += sy/2*1.5;
        vec2_set(projectile->v, 0, 0.25);
    }

    // GAME PHYSICS
    for (int32_t i = 0; i < v->projectiles_count; ++i) {
        struct ci_projectile *projectile = &v->projectiles[i];
        float dt = 1/60.0;
        projectile->p[0] = projectile->p[0] + projectile->v[0] * dt;
        projectile->p[1] = projectile->p[1] + projectile->v[1] * dt;
    }

    // RENDER STUFF
    {
        struct texpack_region *region = &v->regions[0];
        float unit_pixels = 2.0/platform_height(api);
        float sx = region->w * unit_pixels;
        float sy = region->h * unit_pixels;
        vec2 p_a = {v->p[0] - sx/2 , v->p[1] + sy/2};
        vec2 p_b = {v->p[0] + sx/2 , v->p[1] + sy/2};
        vec2 p_c = {v->p[0] - sx/2 , v->p[1] - sy/2};
        vec2 p_d = {v->p[0] + sx/2 , v->p[1] - sy/2};
        vec4 tint = {1,1,1,1};
        gl_particles_add_ex2(&v->particles, &v->regions[0], p_a, p_b, p_c, p_d, tint,
```

```
make: *** [Makefile.comm
Setting up watches.  Bew
Watches established.
./src/game/ CREATE 4913
make in .
mkdir -p build/test/src/
gcc -MD -MP -c -D_POSIX_
msse4.2 -maes -O3 -Wextr
gnedness -Wformat-overfl
-Isrc/core -Isrc/deps_sr
/core -Isrc/game -Isrc/d
ched_include src/game/ci
src/game/ci_view.c: In f
src/game/ci_view.c:167:1
            float sx =
                  ^~
cc1: all warnings being
make: *** [Makefile.comm
Setting up watches.  Bew
Watches established.
./src/game/ CREATE 4913
make in .
mkdir -p build/test/src/
gcc -MD -MP -c -D_POSIX_                                        fPIC -std=c11 -g -
msse4.2 -maes -O3 -Wextr                                        at-y2k -Wformat-si
gnedness -Wformat-overfl                                        Isrc/platform/api
-Isrc/core -Isrc/deps_sr                                        platform/api -Isrc
/core -Isrc/game -Isrc/d                                        /spine-c/build/pat
ched_include src/game/ci
g++ -Wl,-rpath build/tes                                        -o build/test/tmp/
libgame.so build/test/sr                                        /game/game_utils.o
 build/test/src/game/utf                                        est/src/game/tilem
ap.o build/test/src/game                                        e/log_utils.o buil
d/test/src/game/master_v                                        .o build/test/src/
game/srgb_test.o build/t                                        cg_basic.o build/t
est/src/game/gl_blend.o                                         m.o build/test/src
/game/glshapes.o build/t                                        est/src/game/gl_ut
ils.o build/test/src/gam                                        game/smb_character
.o build/test/src/game/smb_death.o build/test/src/game/smb_die_fx.o build/test/src/game/smb_editor.o build
/test/src/game/smb_editor_inputs.o build/test/src/game/smb_inputs.o build/test/src/game/smb_level.o build/
test/src/game/smb_physics.o build/test/src/game/smb_render.o build/test/src/game/smb_spine.o build/test/sr
c/game/smb_view.o build/test/src/game/smb_audio.o build/test/src/game/ci_view.o build/test/src/game/pb_rec
order.o build/test/src/game/pb_inputs.o build/test/src/game/pb_render.o build/test/src/game/pb_view.o buil
d/test/src/game/pb_audio.o build/test/src/game/pb_game.o build/test/src/game/pb_worldmap.o build/test/src/
game/pb_spine.o build/test/src/game/pb_profile.o build/test/src/game/pb_fireworks.o build/test/src/game/sm
b_box2d.o build/test/src/game/box2d_dbg_draw.o
mv build/test/tmp/libgame.so build/test/bin/libgame.so
```

w00tnes: audio works now, yes
Happy_Pip: why c and not c++ or c# ?
Happy_Pip: fair enough
l_39a: What does BOMBY stand for?
Happy_Pip: font is perfect
s3bastians: Hello
l_39a: What is the plan for the game? Are you making it up as you go along, or you already have a clear goal?
l_39a:
l_39a: Didn't you forget /misc when copying into bundle?
l_39a:
maxi_gangster: what are you programming?
maxi_gangster: I made a Matrix Program in C but it was too easy for me. I only made a random number and printed it out to the console and i made the text green with system("color 2");
maxi_gangster: it was just a do while loop
maxi_gangster: yes
maxi_gangster: i wanted to do something harder
l_39a: Will the call to finger_read_next call a similar native routine in the target OS, or is your platform layer separately populating an event queue that you cheaply poll?
l_39a: Why do you keep separate buffers per finger?
l_39a: Doesn't it make the input logic a bit weird, for example with the thumb stick, can it only be controlled by finger 0?
PikkInTheFace: i feel like big c/c++ projects are hard to grasp... got any tips?
basicknowledge: EWW THAT COLOR
l_39a: How long will you still stream tonight?
l_39a: Well Played
l_39a: Good progress in 2 hours

Fig. 1: In live-streamed programming, a developer working on a programming activity shares a live video of their work with other developers, who may interact through chat to ask questions, suggest alternatives, and help find defects.

***Abstract*—In live-streamed programming, developers broadcast their development work on open source projects using streaming media such as YouTube or Twitch. Sessions are first announced by a developer acting as the streamer, inviting other developers to join and interact as watchers using chat. To better understand the characteristics, motivations, and challenges in live-streamed programming, we analyzed 20 hours of live-streamed programming videos and surveyed 7 streamers about their experiences. The results reveal that live-streamed programming shares some of the characteristics and benefits of pair programming, but differs in the nature of the relationship between the streamer and watchers. We also found that streamers are motivated by knowledge sharing, socializing, and building an online identity, but face challenges with tool limitations and maintaining engagement with watchers. We discuss the implications of these findings, identify limitations with current tools, and propose design recommendations for new forms of tools to better supporting live-streamed programming.**

***Index Terms*—screencasting, social coding, pair programming.**

978-1-7281-0810-0/19/$31.00 ©2019 IEEE

I. Introduction

Programming has long been in part a social activity, as developers share knowledge and collaborate on projects using platforms ranging from bulletin boards and mailing lists to GitHub, StackOverflow, and Twitter [1]–[8]. Recently, a new form of collaboration has emerged in which developers live-stream their work developing open source software [9], inviting other developers to join a programming session and watch while engaged in activities such as writing code, debugging, and refactoring. In this paper, we refer to this form of collaboration as *live-streamed programming*. Live-streamed programming sessions are often first announced by a developer, sharing a date and time at which a session will begin using social media such as Reddit and Twitter. The announcement may also include a link to the session, hosted on a video streaming platform such as YouTube or Twitch. During the session, developers join, watch, and interact, using chat with the streaming developer to ask questions, offer suggestions, and warn about potential defects (Figure 1).

Live-streamed programming is increasingly common within open source, with thousands of developers streaming their work and tens of thousands watching [10]. Streamers use communities such as Reddit[1] and Shipstreams[2] to connect with watchers. Developers write about their personal experiences with live-streamed programming, often comparing it to pair programming [11]–[13].

However, many questions remain about the characteristics of live-streamed programming and the motives of its participants. In this paper, we explore the nature of live-streamed programming as a new form of social coding and collaboration between developers. Specifically, we focus on four research questions:

RQ1 What are the characteristics of live-streamed programming?

RQ2 To what extent is live-streamed programming a form of pair programming?

RQ3 What motivates developers to host live-streamed programming sessions?

RQ4 What challenges do developers face in hosting live-streamed programming sessions?

To answer these questions, we conducted an exploratory study in which we watched 20 hours of live-streamed programming videos of seven developers (*streamers*) working on a variety of software development activities and interacting with other developers (*watchers*). We then sent a short survey to the seven streamers and received five responses with insight into their motivations and the challenges they faced.

Our results suggest that live-streamed programming shares some characteristics and benefits with pair programming, but also differs in several key ways. Watchers interact with streamers less frequently and show less commitment. We identified two forms of challenges which developers face. First, streamers struggled to maintain engagement with watchers while working on development activities. Second, watchers were not able to easily explore the codebase and quickly onboard with the task streamers were working on, limiting their ability to collaborate effectively with streamers. Based on these findings, we propose several design recommendations for how tools can more effectively support the live-streamed programming workflow.

II. Background

Social coding [4], [14] offers opportunities for developers to collaborate and share artifacts across organizations, teams, and individuals. Software companies and individual developers use platforms such as Github and GitLab to host open source projects and enable the creation of communities [15], [16]. Developers report that they enjoy the process of open sourcing their software and find it an effective way to improve their technical skills [1]. Researchers have studied how developers discuss and evaluate code contributions on platforms such as Github [7], identifying challenges and best practices regarding code review [8]. Developers use communities such as StackOverflow to share knowledge and help each other by answering programming-related questions [2]. Social coding also occurs through *Screen casting*. Screencasting differs from live-streamed programming in that developers work solo and then post their video for asynchronous consumption. It is often used to document and share knowledge via online video sharing platforms such as YouTube, helping developers to promote themselves by building an online identity [17].

Pair programming is a social and collaborative activity in which two developers sit together at one computer, collaborating on designing, coding, debugging, and testing software. The collaboration between the two developers occurs by each taking two roles interchangeably. The first role is the *driver*, whose responsibility is to act, including writing code, debugging, and testing. The *navigator* is responsible for observing the driver and suggesting strategies, alternative solutions, and possible defects [18]. Several studies have investigated the effectiveness of pair programming in both industrial and educational contexts [19]. Pair programming has been found to enable students to produce better programs [20] and to achieve better grades [21]. In industrial settings, studies suggest that pair programming provides benefits over traditional solo programming. Professional developers report creating higher quality code, inserting fewer defects [22], and an improvement in team communication [23] when practicing pair programming. One form of pair programming is mob programming [24], in which a whole development team works at a single computer. The role of the driver is rotated among team members every 5-15 minutes. This practice has been found to increase team productivity and encourage face to face communication [25].

Distributed pair programming is a form of pair programming in which the driver and the navigator are not collocated and work remotely [26]. For distributed pair programming to have the same benefits as collocated pair programming, tools that support “cross-workspace information infrastructure” must exist [27] to ensure that the driver and navigator communicate effectively [28]. Several tools have been proposed to support distributed pair programming [29]. For example, Saros [30] and XPairtise [31] are Eclipse plugins that help developers to communicate during the pair programming session. Saros is a flexible tool for distributed pair programming, providing a synchronized workspace and visualizations to enhance awareness between the driver and the navigator. XPairtise is built to support students’ collaboration by offering classroom support capabilities to add programming and group assignments.

Live coding is a social coding practice in which the goal is to perform and demonstrate in front of an audience, often constructing small programs with 20 to 50 lines of code [32]. Live programming often occurs to produce electronic music in front of an audience [33]–[35] and to teach programming to students during lectures [36], [37]. Live-streamed programming differs from live coding in that it shows developers engaged in software development activities on projects that are intended to be production quality software (e.g., Shipstreams).

Prior studies of live-streamed programming have explored

[1]https://www.reddit.com/r/WatchPeopleCode

[2]https://shipstreams.com/

TABLE I: A summary of the ten sessions. The projects' number of contributors and lines of code (LOC) indicate the size and level of activity of each project. Each project URL is prefixed with https://github.com/.

Session	Streamer	Main Software Development Activity	Contributors	LOC	Project URL	Duration Obs.
1	D1	Implementing Features	5	385 Python	eleweek/SearchingReddit	2:57:35
2	D1	Implementing Features	5	385 Python	eleweek/SearchingReddit	1:49:00
3	D2	Implementing Features	1	423 C++	benhoff/face-recognizer-gui	1:25:28
4	D2	Implementing Features	1	423 C++	benhoff/face-recognizer-gui	2:02:36
5	D3	Code Refactoring	41	32.6K Rust	graphql-rust/juniper	2:07:00
6	D4	Responding to Issues and Pull Requests	299	90.3K JavaScript	pouchdb/pouchdb	2:37:00
7	D5	Implementing Features	Unknown	Unknown C	Unknown	2:26:30
8	D6	Debugging	37	8.4K JavaScript	wulkano/kap	1:21:06
9	D6	Debugging	50	8K JavaScript	buttercup/buttercup-desktop	1:31:24
10	D7	Code Refactoring	24	14K JavaScript	noopkat/avrgirl-arduino	1:29:37

its educational implications. Haaranes [38] observed one streamer during three sessions building a game and proposed teaching methods that expose computer science students to larger software projects. Faas et al. extended this work with two studies [39], [40] involving observations and interviews of streamers on Twitch. This work revealed the mentoring relations that occur in live-streamed programming and its potential for use as a learning platform. In this paper, we build upon this work by exploring live-streamed programming as a form of social coding and collaboration. In particular, we investigate its characteristics, how it differs from pair programming, and the motivations and challenges of its participants.

III. METHOD

To answer our research questions, we first selected 20 hours of live-streamed programming sessions. To identify these sessions, one approach would be to wait for streamers to announce live-streamed programming sessions through social platforms and then join to observe in real time. However, this limits observations to new sessions. As we did not wish to actively participate, this offered no benefits. We thus chose to search for archived sessions on platforms such as YouTube, Twitch, and Reddit. We used two inclusion criterion to select sessions. First, each session should show a streamer engaged in a software development activity, such as writing, debugging, or refactoring code. Second, each session should show a streamer and watchers interacting via voice or text, and these interactions should be accessible via the video or chat history. We searched for sessions that satisfied these criteria and selected ten, all of which were hosted on YouTube. We did not include videos hosted on Twitch and Reddit, as these videos may not be archived. For examples, Twitch only archives videos for 14 days [3].

The 20 hours we selected included ten videos by 7 streamers (six male, one female; D1-D7) working on both personal open source projects (D1, D2, D5, D7) as well as contributing to popular open source projects (D3, D4, D6). Table I summarizes the sessions. We watched each of these sessions, identifying common activities among the streamers and watchers. We then compared our observations with descriptions of pair programming reported in prior work [18].

To further understand the challenges and motivations of streamers, we sent a brief two question survey to the seven streamers, focusing on their motivations and challenges. Five responded (D1-D5). The sessions links and the data we collected from the sessions are publicly available[4].

IV. RESULTS

A. What are the characteristics of live-streamed programming?

Our observations of the ten live-streamed programming sessions revealed common characteristics in how developers *advertise*, *start*, *plan*, *use*, and *end* sessions. Streamers first *advertised* their sessions through announcements on Twitter (D3, D4, D6, D7) or Reddit (D1, D2, D5), including the expected time of the streaming session and a link to the session. All except session 9 had a posted announcement before the start of the stream.

Each session *started* with developers showing their face and giving an introduction and background for the session. This lasted an average of 9 minutes (range 1-18 minutes). Streamers educated watchers about the codebase, technologies used, and the purpose of the session. All sessions were streamed from a private space except session 10, which occurred in a public coffee shop.

Streamers often began their work by offering a general *plan* of what they would do. D5 stated that he would build a game but he "*do[es] not know what the game exactly will be*". D4 streamed the process of debugging issues reported in open source projects he maintained. Not all the issues were resolved during the session, as he stopped while debugging some and moved on to others. This left the live-streamed session without a clear goal, leading some watchers to ask when the session would end.

After finishing the introduction, the streamers *used* the session to engage in development activities, including implementing features, debugging, refactoring, and searching for documentation and StackOverflow posts. Watchers interacted with streamers during these activities by asking questions and helping the streamer with the task at hand, which sometimes prompted new behavior by the streamers. For example, when watchers posted a library to use or a link to read in the

[3]https://help.twitch.tv/s/article/videos-on-demand,

[4]https://github.com/Alaboudi1/live-streamed-programming

chat, streamers almost always stopped their current activity to check what the watchers suggested. Streamers (D1, D4, D6) were interrupted by people in their physical environment who brought drinks, talked with them, or rang their doorbell. While most streamers waited until the end of their session to commit their code changes to their version control system, D1, D3, and D4 continually pushed their changes throughout the sessions, enabling watchers to run and test them.

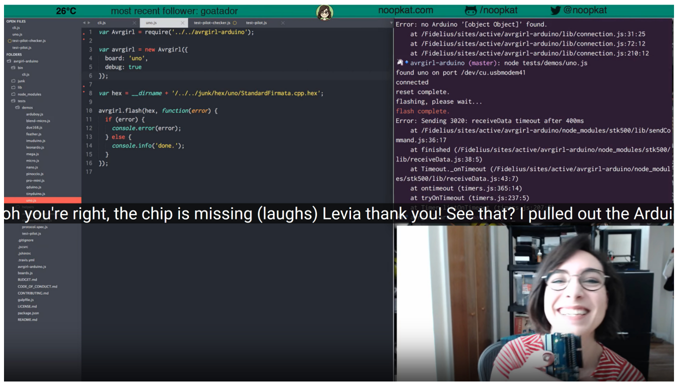
While streamers were engaged in development activities, watchers continually joined and left throughout. When they first joined, watchers sometimes communicated with the streamer or with other watchers through chat. However, most watchers remained passive participants. For example, session 1 had over 50 watchers but only 12 who sent at least one chat message.

Streamers often *ended* the live-streamed programming session by giving a demo of the work they had completed. They also pushed their final code changes to a public repository for the watchers to view. Finally they often concluded by stating the tentative time and date, if any, of their next session.

B. *To what extent is live-streamed programming a form of pair programming?*

In **pair programming**, there are two distinct roles. The role of the driver is to take action, while the navigator supports and guides the driver [18]. In the live-streamed programming sessions we observed, the behavior of the streamers and watchers generally followed these roles. Streamers wrote code, debugged, and designed, while watchers observed the streamer, helping debug, suggesting alternative solutions, and offering tips. However, effective pair programming requires the driver and navigator to interact at least once every 45 to 60 seconds [18]. In our observations, we did not find that the streamers and watchers collaborated at this level of frequency. This made live-streamed programming different from pair programming in two key ways. First, streamers did not actively solicit input from watchers. Throughout the sessions, streamers educated watchers about the project and explained design decisions and technology choices they made. When watchers helped streamers debug or find better tools and solutions, streamers thanked the watchers, as they were not expecting such a contribution (Figure 2). Second, watchers exhibited weaker focus and commitment to the task at hand. Watchers joined and left throughout the sessions, inspecting and working on other parts of the code if it was available to them, and interrupted the streamer by asking questions that were not relevant to the current task or which had been answered before. Figure 3 lists examples of interactions between the streamer and watchers from session 3.

Studies have found that pair programming offers several benefits, such as producing high-quality code, shortening development time, facilitating knowledge sharing, and creating joy in the work environment [18]. The observations and survey responses suggest that live-streamed programming may offer some of these same benefits. We were able to identify seven instances in which watchers improved the quality of their code,


26°C
most recent follower: goatador
noopkat.com
/noopkat
@noopkat
OPEN FILES
cli.js
uno.js
test-pilot-checker.js
test-pilot.js
FOLDERS
avrgirl-arduino
oh you're right, the chip is missing (laughs) Levia thank you! See that? I pulled out the Ardui

```
var Avrgirl = require('../../avrgirl-arduino');

var avrgirl = new Avrgirl({
  board: 'uno',
  debug: true
});

var hex = __dirname + '/../../junk/hex/uno/StandardFirmata.cpp.hex';

avrgirl.flash(hex, function(error) {
  if (error) {
    console.error(error);
  } else {
    console.info('done.');
  }
});
```

```
Error: no Arduino '[object Object]' found.
    at /Fidelius/sites/active/avrgirl-arduino/lib/connection.js:31:25
    at /Fidelius/sites/active/avrgirl-arduino/lib/connection.js:72:12
    at /Fidelius/sites/active/avrgirl-arduino/lib/connection.js:210:12
avrgirl-arduino (master): node tests/demos/uno.js
found uno on port /dev/cu.usbmodem41
connected
reset complete.
flashing, please wait...
flash complete.
Error: Sending 3020: receiveData timeout after 400ms
    at /Fidelius/sites/active/avrgirl-arduino/node_modules/stk500/lib/sendCo
mmand.js:36:17
    at finished (/Fidelius/sites/active/avrgirl-arduino/node_modules/stk500/
lib/receiveData.js:38:5)
    at Timeout._onTimeout (/Fidelius/sites/active/avrgirl-arduino/node_modul
es/stk500/lib/receiveData.js:43:7)
    at ontimeout (timers.js:365:14)
    at tryOnTimeout (timers.js:237:5)
```

Fig. 2: A streamer (D7) thanked a watcher for finding a defect.

reporting three defects and helping debug another four. D1 stated during session 2 that he would not have discovered a defect a watcher reported, as it was related to an edge case that he was not aware of: "*Thank you! I would not have thought about the accented character*". Another streamer (D5) tested multiple incorrect hypotheses about the cause of a defect before a watcher suggested a correct hypothesis.

STREAMER: [After reading the error] I think I know why, but before I say it I want to confirm it.

WATCHER: Did you forget /misc when copying into a bundle?

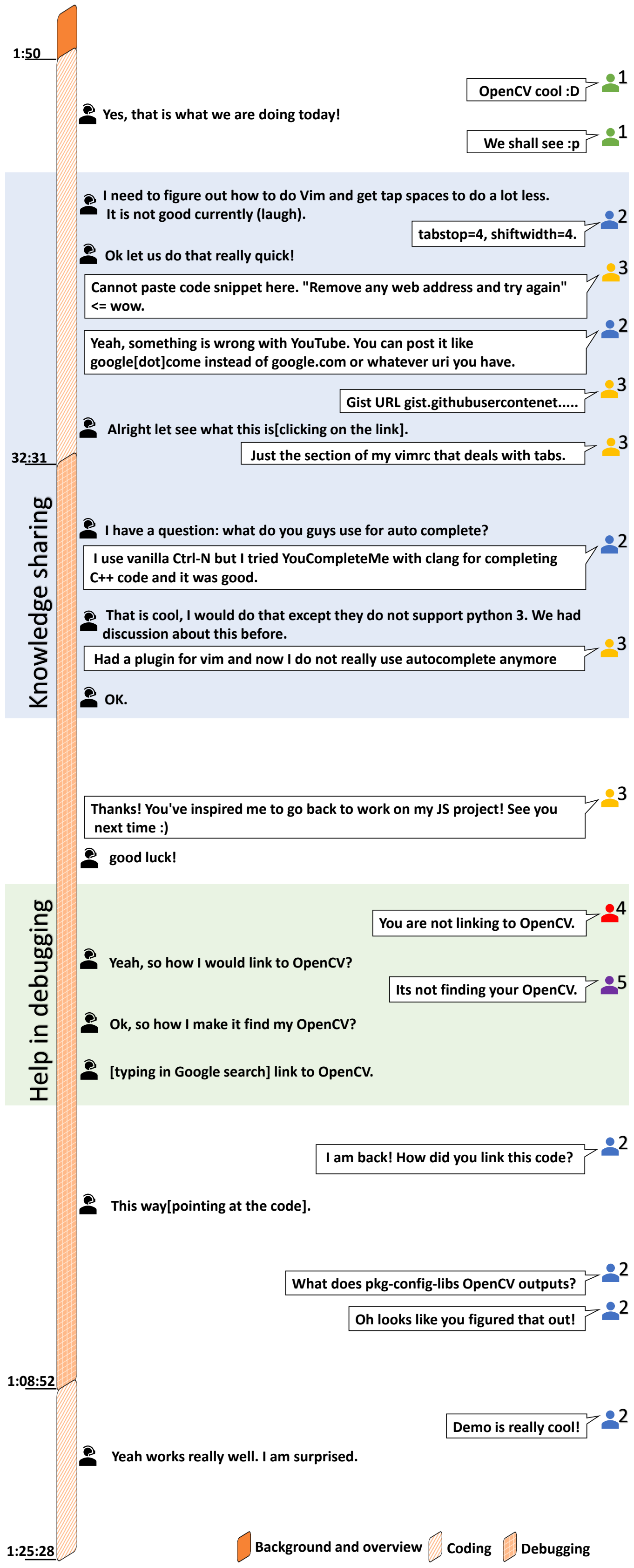
STREAMER: [Stopped debugging and started reading the chat] I do not think so, but thanks! Let's see.

STREAMER: [After testing the watcher's suggested hypothesis] Bravo, you are right! I forgot to do that, thank you!

We observed 38 instances in which streamers and watchers shared tips, explaining technical topics and offering alternative technology choices to each other. D4 was explaining his git workflow to the watchers, and one of the watchers pointed out a better workflow that he did not know. "*[Reading the tip message from the chat] Oh thank you, Jake! Let us try it out. [He tries it.] OK, that is a lot faster than [the] three steps that I was doing before, thank you! That is going to improve my efficiency a lot*". D3 was working on refactoring an existing codebase in an open source library, where the maintainer of the library was among the watchers. Both the streamer and the watcher (maintainer) asked and answered each others' questions while the streamer was working. When the streamer finished working on the project, the watcher wrote "*it has been very useful, learned new things, thanks!*".

One advantage of pair programming that we did not observe was shortening development time. Two developers (D2, D4) explicitly stated in their sessions that they usually take less time doing the same job offline. "*I really [am] only keeping doing this because people say, hey, I like the thing you are doing. Otherwise, I really would not do it, because I would code faster without [it]*" - D2. One contributing factor might be a lack of effective tooling, which we discuss in RQ4 below.

Mob programming is a form of pair programming in which more than two developers collaborate and rotate the role of



OpenCV cool :D

Yes, that is what we are doing today!

We shall see :p

I need to figure out how to do Vim and get tap spaces to do a lot less. It is not good currently (laugh).

tabstop=4, shiftwidth=4.

Ok let us do that really quick!

Cannot paste code snippet here. "Remove any web address and try again" <= wow.

Yeah, something is wrong with YouTube. You can post it like google[dot]come instead of google.com or whatever uri you have.

Gist URL gist.githubusercontenet.....

Alright let see what this is[clicking on the link].

Just the section of my vimrc that deals with tabs.

I have a question: what do you guys use for auto complete?

I use vanilla Ctrl-N but I tried YouCompleteMe with clang for completing C++ code and it was good.

That is cool, I would do that except they do not support python 3. We had discussion about this before.

Had a plugin for vim and now I do not really use autocomplete anymore

OK.

Thanks! You've inspired me to go back to work on my JS project! See you next time :)

good luck!

You are not linking to OpenCV.

Yeah, so how I would link to OpenCV?

Its not finding your OpenCV.

Ok, so how I make it find my OpenCV?

[typing in Google search] link to OpenCV.

I am back! How did you link this code?

This way[pointing at the code].

What does pkg-config-libs OpenCV outputs?

Oh looks like you figured that out!

Demo is really cool!

Yeah works really well. I am surprised.

Fig. 3: Excerpts from session 3 which offer examples of the interactions between the streamer (left) and watchers (right). These excerpts illustrate knowledge sharing between the streamer and the watchers, watchers helping in debugging, and watchers joining and leaving throughout the session.

driver between them. We did not observe any instances of a streamer and watchers rotating their roles.

C. What motivates developers to host live-streamed programming sessions?

From our observations of the sessions as well as the survey responses, we identified three motives for live-streamed programming: *knowledge sharing*, *socializing and work enjoyment*, and *building an online identity*.

We found **knowledge sharing** to be a common activity in live-streamed programming, confirming findings from prior studies [38]–[40]. We identified several instances in which streamers and watchers explained to each other design recommendations and choices, alternative tools and libraries, and tips they found to increase their productivity. D3 stated at the beginning of a live-streamed programming session that he does the recording for people who " *want to learn or see the language [Rust] being used in a more advanced way*". He further expanded on this motivation in his survey response:

"*I find it very rewarding to hear that others are learning from what I'm putting out there. I've always enjoyed teaching, and this medium seems to work very well for a lot of people.*"

D4 reported in his survey response that what motivated him is "*to show"a daily in the life"of an open-source maintainer*". He pointed us to a blog post that he wrote about the struggles open source software maintainers of popular library experience day to day [41]. During his session, he went through 69 separate Github issues and pull requests opened by the community, showing how he answers these issues and reviews code written by other developers.

D1 indicated that, besides his live-streamed programming sessions, he also joins other developers' sessions, collaborating with them and assisting novices while developing software.

"*My favorite aspect of streaming was collaboration. I'd even say that more than my own streaming I enjoyed helping other people on their streams more - especially novices.*"- D1

Socializing and work enjoyment was another key motive. D5 stated that what made him start to host live-streamed programming sessions was because he "*enjoyed watch[ing] others coding [...] and thought it would be fun as an experiment to try to do that too*". D3 stated that socializing during live-streamed programming session made him enjoy his development work to a degree that he would work for 5-6 hours continuously, and that he often uses it to motivate him to work on his research.

"*I genuinely enjoy the experience of programming with other people. It feels like we're solving something together, and having them there makes the entire experience more fun. [...] That's amazing! Often I will choose projects that tie into my research too, so it's even useful work.*" - D3

Watchers expressed enjoyment in the process of sharing, watching the development workflow, and collaborating with streamers. One watcher expressed enjoyment after suggesting an idea to the streamer during session 1. "*This is so cool... :) watching others code and contributing with ideas*". Another watcher enjoyed the idea of watching another developer while coding. "*It is really like hardcore watching [another] hardcore coding like this. I love it!*" (session 2).

A third motive was **building an online identity**. D1 stated that a recruiter from Google had contacted him because of his streaming activity. D2 has a similar experience where he "*had multiple new job opportunities that came out of streaming online*". D3 expressed no interest in job opportunities, as he was still in graduate school, but stated that "*with the videos I think there's now a lot more people who know who I am*". D4 pointed out that although his online identity had not grown because of his streaming activity, he was happy that he inspired another developer to become a streamer (D7), who later landed a job at Microsoft because of her streaming activity [11].

D. What challenges do developers face hosting live-streamed programming sessions?

Our observations and survey responses revealed two key challenges that streamers and watchers experienced.

Tools limitations. Our observations suggest that watchers sometimes help streamers while debugging, and thus may need access to the source code to inspect and test their hypotheses. These needs were not directly supported by any of the development tools used by the streamers. For example, a watcher was trying to help D7 while debugging, but the interaction was slow and consisted of "*Are you referring to this line?*". This largely was due to a lack of direct access to the code. To answer a question, he needed first to write the question down in chat and wait for the streamer to read it and respond, which sometimes then required a request for clarification. D7 referred to this interaction as "*slow, delayed pair programming*" while the watcher felt guilty for not being able to help: "*oh my gosh I am terrible*".

D1 stated in the survey that the lack of tools which "*remove some friction with figuring out what the streamer is doing right now and allow more collaboration*" created a challenge in maintaining productive live-streamed programming sessions with watchers.

"*I really wish there was a way of letting viewers browse your code independently, and maybe even a way of quickly recreating the same dev environment in some way.*" - D1

In addition to giving watchers the ability to explore the code independently, watchers may need help in understanding the codebase and the current task of the streamer, particularly when they join a session late. D2 reported that existing tools lack support for giving a quick overview of the project and its architecture for watchers.

"*I think the hardest thing the viewers would have was understanding the architecture of the code base, or how things fit together. So if you can figure out how to keep track of the problem that you are working on, visually represent the code base/changes made against the task, and how that code fits together, you would have a good visual editor.*" - D2

Maintaining engagement. Writing and debugging code are cognitively demanding tasks which may often require the developer's full attention. Streamers often stayed silent while they were thinking about what might cause a defect in their code. This may create a disconnect between the streamer's and watchers' mental models, leaving watchers less engaged in the current task.

"*I think the hardest thing is to make sure you keep the viewers engaged. Don't just code without speaking – you need to voice your thoughts, and ask the people watching the same questions you'd normally mull over in your head while programming. Otherwise they won't learn anything as it'll be an entirely passive experience. And I don't think people are generally used to speaking their thoughts out loud while programming!*" - D3

Another streamer reported that coding while a camera was recording impacted his ability to maintain watchers' engagement.

"*While coding, there are many other thoughts that need to be managed. You literally have a camera on you while you are coding, and that can greatly affect my focus. Likewise, once I am concentrating on some code - I may forget about the camera and not explaining my thought-process clearly*" - D5

V. Limitations and threats to validity

Our study has several important limitations and potential threats to validity. One potential threat to external validity is how representative the ten live-streamed programming sessions that we selected were. We included sessions based on the development activities and the presence of interactions between streamers and watchers. However, our selected sessions were also diverse in programming languages, projects size (ranging from 385 to 90.3K lines of code), and the number of developers contributing to the projects. Another potential threat to the generalizability of our results is the size of our sample. In our study, we observed a total of ten sessions which spanned 20 hours of activity by 7 streamers. Prior work [42]–[45] which involved analysis of developers working in various development activities had on average 6.6 developers (range 3-10) and an average of 8 hours of videos (range 5-15).

A potential threat to internal validity is related to our qualitative data analysis. As our focus was to identify the nature of live-streamed programming work rather than characterize the frequency of specific practices or characteristics, we employed a lightweight data analysis and did not code most events for their frequency.

VI. Discussion

Live-streamed programming is a new form of collaboration in which streamers invite watchers to observe them perform programming work. Our observations revealed that developers' roles, interactions, and benefits have similarities to pair programming. However, live-streamed programming differs in two important ways. Streamers and watchers interactions were less frequent, and watchers were less committed to the task at hand, joining and leaving throughout the session. Streamers were motivated by the opportunity to use their work as an educational resource and as a means to promote themselves, confirming findings in prior studies [17], [38]–[40]. One additional motive we observed was that both streamers and watchers enjoyed the process of streaming and watching coding activities. D3 reported that streaming helped him to enjoy working for hours for different projects, including his academic research work. Further research on this motive may help to better reveal how live-streamed programming might be used create an enjoyable and stimulating working environment, assisting developers in enjoying daunting coding activities and potentially increasing their productivity.

One unique characteristic of live-streamed programming is that, unlike the navigator in pair programming, watchers may or may not contribute to the task at hand and may join and leave throughout the session. Current tools that support distributed pair programming such as Saros [30], XPairtise [31], and Visual Studio Live Share [46] were not designed to support this form of work. For example, they do not offer support for watchers who join late in gaining a holistic view of what the streamer is working on and planning for the rest of the session. Further, existing pair programming tools require investment and commitment by the watcher to install and configure the development environment to run them. Online development environments such as Codenvy[5], Codeanywhere[6], and Cloud9[7] may require less setup. But they may offer development tools and workflow that differ from what the streamers use locally. They also lack other features to support the workflow of live-streamed programming, discussed below.

The lack of effective tooling for live-streamed programming has already motivated some open source developers to invent new tools that address some of the challenges we identified. During the time in which this study was conducted, an open source extension for VS Code[8] was introduced which allows watchers to refer to a location in the source code via chat, which results in an in-editor visualization for the streamer. While addressing one key aspect of streamer and watcher collaboration for one specific development environment, there remain additional challenges to be addressed such as how to enable watchers to explore the code independently, how to help them with onboarding with the streamer's development work, and how to help streamers maintain watchers' engagement. We discuss design implications for tools in the following section.

VII. Design Implications

Our study revealed that streamers sometimes struggle in *maintaining engagement* with watchers, while watchers need tools that help them *explore code* independently and help *onboard* to the streamer's project. In this section, we propose a set of design recommendations for addressing these needs.

A. Maintaining Engagement

Instead of relying exclusively on the streamer to continually remember to keep watchers engaged while immersed in development activities, watchers might instead help in quickly notifying the streamer of their needs. For example, watchers may want to inform the streamer to think aloud to make them follow along and avoid making them speculate as to why she acted as she did. We call these notifications from watchers *Fast interactions*.

Fast interactions are a way for watchers to communicate common needs to the streamer quickly. Instead of writing a text message to remind the streamer to think aloud, watchers might click a button to notify the streamer to talk through what she is currently doing. Watchers may need access to the latest version of the code or to increase the font size to make the code in the streamer's editor more easily readable. Watchers may also want to give a sign of support to inform the streamers that they are following along. Faster interactions might be triggered by either the streamer or the watcher. This form of communication may also encourage more watchers to actively collaborate by lowering the barriers to participate.

B. On-demand Code Exploration

Our findings suggest that watchers may benefit from being able to explore the code independently from the streamers. However, the latest version of the codebase may not always be accessible for the watchers or may require the installation of several dependencies and modification to the environment variables to build the project and run it. Such barriers increase the cost of code exploration for the watchers and may decrease their engagement with the streamer. This suggests that a tool to support live-streamed programming should offer on-demand code exploration for watchers. Additionally it should not require any tool or development environment set-up for either the streamer or for the watchers.

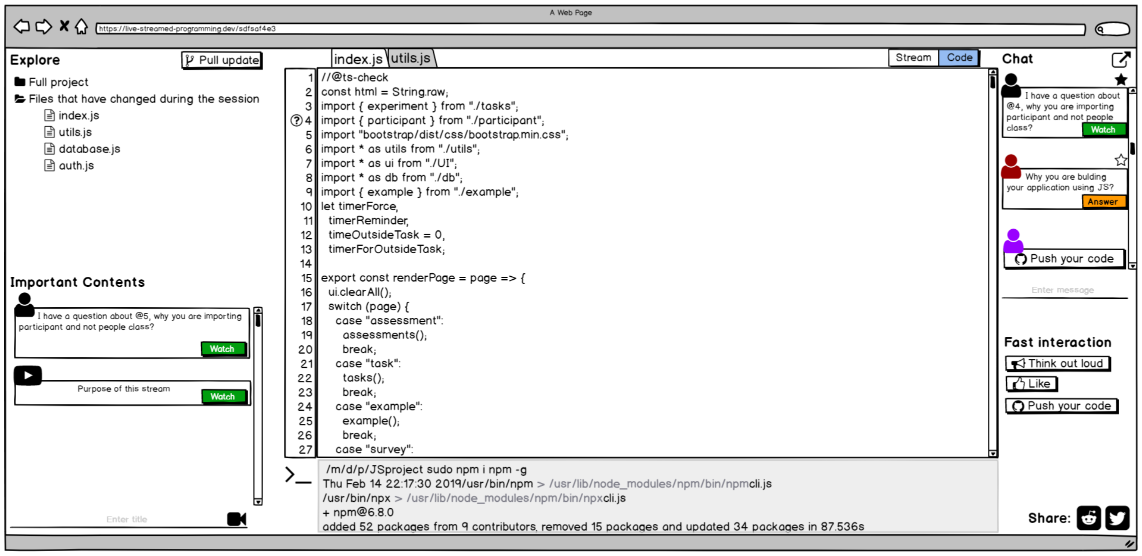
One approach may be to build a web application that connects to the streamer's project repository (e.g., Github) and offers the necessary environment for the watchers to build and run the software. Watchers may also need to know what files have been changed recently so that they can only focus on these files instead of the entire project.

C. Fast Onboarding

One difference between pair programming and live-streamed programming is that, unlike navigators, watchers join and leave throughout the session. Watchers who join later may have questions that have already been answered about the purpose of the project, previous design decisions, and the plan for the rest of the session. To help watchers quickly answer these questions and get up to speed, tools might offer *content linking* that help watchers quickly traverse the history of the session to find answers for their questions.

Tools such as chat.codes offer the ability to link between code fragments and natural language descriptions in "asynchronous settings where one user writes an explanation for another user to read and understand later" [47]. Our proposed feature for content linking extends this idea to consider linkages between video segments, code fragments, and natural language text in the chat. Streamers might indicate video segments in the stream that are of importance to watchers who came late. These video segments may include the introduction

[5]https://codenvy.com
[6]https://www.codeanywhere.com
[7]https://aws.amazon.com/cloud9
[8]https://github.com/clarkio/vscode-twitch-highlighter



I have a question about @5, why you are importing participant and not people class?

Purpose of this stream

```
//@ts-check
const html = String.raw;
import { experiment } from "./tasks";
import { participant } from "./participant";
import "bootstrap/dist/css/bootstrap.min.css";
import * as utils from "./utils";
import * as ui from "./UI";
import * as db from "./db";
import { example } from "./example";
let timerForce,
  timerReminder,
  timeOutsideTask = 0,
  timerForOutsideTask;

export const renderPage = page => {
  ui.clearAll();
  switch (page) {
    case "assessment":
      assessments();
      break;
    case "task":
      tasks();
      break;
    case "example":
      example();
      break;
    case "survey":
```

```
 /m/d/p/JSproject sudo npm i npm -g
Thu Feb 14 22:17:30 2019/usr/bin/npm > /usr/lib/node_modules/npm/bin/npmcli.js
/usr/bin/npx > /usr/lib/node_modules/npm/bin/npxcli.js
+ npm@6.8.0
added 52 packages from 9 contributors, removed 15 packages and updated 34 packages in 87.536s
```

I have a question about @4, why you are importing participant and not people class?

Why you are bulding your application using JS?

Fig. 4: A tool mockup illustrating our design recommendations

to the session or answers to key questions posed in the chat. Besides video segment linking, watchers who have questions about the codebase should have the ability to link their questions with a code location, helping offer both the streamer and other watchers more context.

D. Tool Workflow

To illustrate how our design recommendations might be embodied in a tool for supporting live-streamed programming, Figure 4 depicts a mockup of a web-based tool for live-streamed programming. Before the live-streaming session begins, the streamer first logs into the tool website and provide links to both the live-streamed programming session and the public repository that hosts the source code. The tool then aggregates access to the session and the code repository in one web page. The streamer may then advertise the session by using the share button or copying the page link and posting it to social platforms. After the session begins, the streamer may work in her preferred development environment and push any updates to the code repository for the watchers to use.

When watchers join the session, they have the option to either watch the live stream or independently explore the codebase, triggered through a toggle button (Stream Code). Invoking the code mode brings up a lightweight development environment containing a code editor, terminal, and file browser. In the file browser, watchers can pull updates from the repository, explore the entire project, or limit the focus to the files that have been recently updated during the session. Watchers may interact with the streamer via chat or fast interaction (bottom right in Figure 4). In the regular chat, watchers can link to a code location in the chat using @. This creates a (?) mark on the gutter at that location so that other watchers can see the presence of discussions by skimming the gutter.

When answering a question posted in the chat, the streamer can mark the beginning of the answer by clicking on (Answer), linking the question to its answer to make it easy for latecomers to find the answer by clicking on (Watch). Both the streamer and watchers may mark important chat questions by clicking on the (☆) and indicate segments of the session where the streamer discusses important topics by clicking on (■◀). This might be used to create a list of important questions and moments for late watchers to explore.

VIII. Conclusion

Live-streamed programming is a form of collaboration in which a streamer synchronously broadcasts their programming work and interacts with watchers. Our results offer insight into its characteristics, why developers use it, and the challenges they face. Our findings suggest that live-streamed programming shares several characteristics and benefits with pair programming, but also differs in ways which make existing distributed pair programming tools less effective. We propose several design recommendations for how future tools can offer more effective support. We believe that such tools may enable more developers to make portions of their development work public and accessible for other developers to observe and collaborate.

Acknowledgments

We would like to thank the survey respondents for their time. This research was funded in part by NSF grant CCF-1703734. The first author is supported in part by a King Saudi University Graduate Fellowship.

References

[1] K. R. Lakhani and E. Von Hippel, "How open source software works:"free" user-to-user assistance," in *Produktentwicklung mit virtuellen Communities*. Springer, 2004, pp. 303–339.

[2] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Conference on Human Factors in Computing Systems*, 2011, pp. 2857–2866.

[3] L. Singer, F. Figueira Filho, and M.-A. Storey, "Software engineering at the speed of light: how developers stay current using twitter," in *International Conference on Software Engineering*, 2014, pp. 211–221.

[4] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *Conference on Computer Supported Cooperative Work*, 2012, pp. 1277–1286.

[5] B. Vasilescu, V. Filkov, and A. Serebrenik, "Stackoverflow and github: Associations between software development and crowdsourced knowledge," in *International Conference on Social Computing*, 2013, pp. 188–195.

[6] A. Bosu, C. S. Corley, D. Heaton, D. Chatterji, J. C. Carver, and N. A. Kraft, "Building reputation in stackoverflow: An empirical investigation," in *International Conference on Mining Software Repositories*, 2013, pp. 89–92.

[7] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: Evaluating contributions through discussion in github," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 144–154.

[8] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, pp. 34–42, jul 2018.

[9] J. Koebler, "Thousands of people are watching this guy code a search engine," 2015, accessed: 2019-01-16. [Online]. Available: https://motherboard.vice.com/en_us/article/pgax4n/thousands-of-people-are-watching-this-guy-code-a-search-engine

[10] C. Bernasconi, "Live programming on twitch is growing fast," 2019, accessed: 2019-06-25. [Online]. Available: https://www.claudiobernasconi.ch/2019/05/29/live-programming-on-twitch-is-growing-fast/

[11] S. Hinton, "Lessons from my first year of live coding on twitch," 2017, accessed: 2019-01-16. [Online]. Available: https://medium.freecodecamp.org/lessons-from-my-first-year-of-live-coding-on-twitch-41a32e2f41c1

[12] A. G. Bell, "Rethinking databases and noria with jon gjengset," 2019, accessed: 2019-06-25. [Online]. Available: https://corecursive.com/030-rethinking-databases-with-jon-gjengset/

[13] S. H. Adam Stacoviak, Jerod Santo, "Live coding open source on twitch," 2018, accessed: 2019-06-25. [Online]. Available: https://changelog.com/podcast/288

[14] A. Begel, R. DeLine, and T. Zimmermann, "Social media for software engineering," in *Workshop on Future of software engineering research*, 2010, pp. 33–38.

[15] L. Hecht and L. Clark, "Survey: Open source programs are a best practice among large companies," 2018, accessed: 2019-03-02. [Online]. Available: https://thenewstack.io/survey-open-source-programs-are-a-best-practice-among-large-companies/

[16] M. Asay, "Who really contributes to open source," 2018, accessed: 2019-03-02. [Online]. Available: https://www.infoworld.com/article/3253948/who-really-contributes-to-open-source.html

[17] L. MacLeod, M.-A. Storey, and A. Bergen, "Code, camera, action: how software developers document and share program knowledge using youtube," in *International Conference on Program Comprehension*, 2015, pp. 104–114.

[18] L. Williams and R. Kessler, *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[19] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," in *International Conference on Software Engineering*, May 2005, pp. 495–504.

[20] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The effects of pair-programming on performance in an introductory programming course," in *Technical Symposium on Computer Science Education*, 2002, pp. 38–42.

[21] N. Nagappan, L. Williams, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the cs1 experience with pair programming," in *Technical Symposium on Computer Science Education*, 2003, pp. 359–362.

[22] W. Cunningham, L. Williams, R. R. Kessler, and R. Jeffries, "Strengthening the case for pair programming," *IEEE Software*, vol. 17, pp. 19–25, 07 2000.

[23] A. Cockburn and L. Williams, "The costs and benefits of pair programming," in *EXtreme Programming and Flexible Processes in Software Engineering*. Addison-Wesley, 2000, pp. 223–247.

[24] J. Dalton, *Mob Programming*. Apress, 2019.

[25] W. Zuill and K. Meadows, "Mob programming: A whole team approach," in *Agile 2014 Conference, Orlando, Florida*, 2016.

[26] P. Baheti, E. Gehringer, and D. Stotts, "Exploring the efficacy of distributed pair programming," in *Conference on Extreme Programming and Agile Methods*, 2002, pp. 208–220.

[27] N. V. Flor, "Globally distributed software development and pair programming," *Commun. ACM*, vol. 49, pp. 57–58, Oct. 2006.

[28] G. Canfora, A. Cimitile, G. Di Lucca, and C. A. Visaggio, "How distribution affects the success of pair programming." *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, pp. 293–313, 04 2006.

[29] B. J. da Silva Estácio and R. Prikladnicki, "Distributed pair programming: A systematic literature review," *Information and Software Technology*, vol. 63, pp. 1–10, 2015.

[30] S. Salinger, C. Oezbek, K. Beecher, and J. Schenk, "Saros: An eclipse plug-in for distributed party programming," in *Workshop on Cooperative and Human Aspects of Software Engineering*, 2010, pp. 48–55.

[31] D. Tsompanoudi, M. Satratzemi, and S. Xinogalos, "Exploring the effects of collaboration scripts embedded in a distributed pair programming system," in *conference on Innovation and technology in computer science education*, 2013, pp. 225–230.

[32] A. Blackwell, A. McLean, J. Noble, and J. Rohrhuber, "Collaboration and learning through live coding (Dagstuhl Seminar 13382)," *Dagstuhl Reports*, vol. 3, pp. 130–168, 2014.

[33] C. Nilson, "Live coding practice," in *Conference on New interfaces for musical expression*, 2007, pp. 112–117.

[34] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised sound*, pp. 321–330, 2003.

[35] T. Magnusson, "Algorithms as scores: Coding live music," *Leonardo Music Journal*, pp. 19–23, 2011.

[36] L. J. Barker, K. Garvin-Doxas, and E. Roberts, "What can computer science learn from a fine arts approach to teaching?" in *Technical Symposium on Computer Science Education*, 2005, pp. 421–425.

[37] C. Chen and P. J. Guo, "Improv: Teaching programming at scale via live coding," in *Conference on Learning at Scale*, 2019.

[38] L. Haaranen, "Programming as a performance: Live-streaming and its implications for computer science education," in *Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 353–358.

[39] T. Faas, L. Dombrowski, A. Young, and A. D. Miller, "Watch me code: Programming mentorship communities on twitch.tv," *Human-Computer Interaction*, vol. 2, pp. 50:1–50:18, Nov. 2018.

[40] T. Faas, L. Dombrowski, E. Brady, and A. Miller, "Looking for group: Live streaming programming for small audiences," in *Information in Contemporary Society*, 2019, pp. 117–123.

[41] N. Lawson, "What it feels like to be an open-source maintainer," 2017, accessed: 2019-02-08. [Online]. Available: https://nolanlawson.com/2017/03/05/what-it-feels-like-to-be-an-open-source-maintainer/

[42] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering.*, vol. 30, pp. 889–903, Dec. 2004.

[43] M. Abi-Antoun, N. Ammar, and T. LaToza, "Questions about object structure during coding activities," in *Workshop on Cooperative and Human Aspects of Software Engineering*, 2010, pp. 64–71.

[44] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering.*, vol. 34, pp. 434–451, Jul. 2008.

[45] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *International Conference on Software Maintenance*, 2009, pp. 157–166.

[46] A. Silver, "Introducing visual studio live share," 2017, accessed: 2019-06-25. [Online]. Available: https://code.visualstudio.com/blogs/2017/11/15/live-share

[47] S. Oney, C. Brooks, and P. Resnick, "Creating guided code explanations with chat.codes," *Human-Computer Interaction.*, vol. 2, pp. 131:1–131:20, Nov. 2018.