Automated Debug & Profiling of AAA Games

UBISOFT
PRESS « F5 » FOR A BETTER EXPERIENCE

Thanks Thomas and the ASE committee for the opportunity to present what we do in such a prestigious conference.

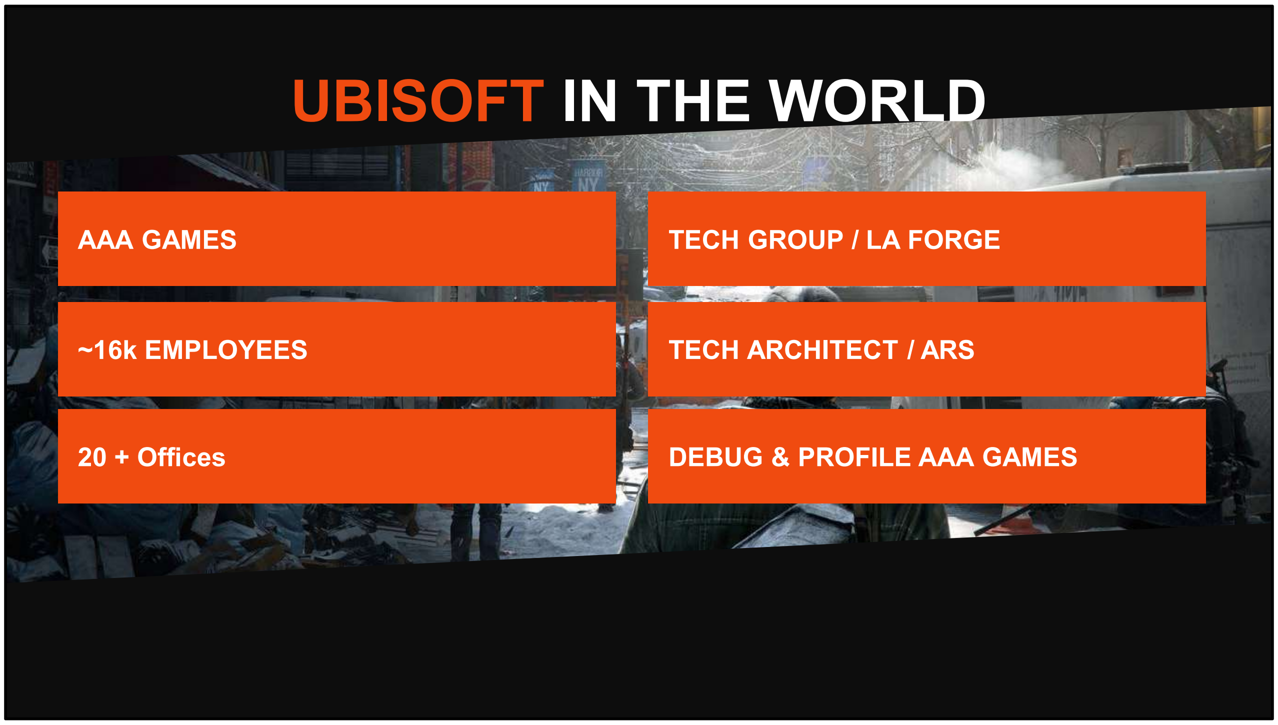
In today's talk, I'll first present what AAA videos games are and, roughly, how they are built. Then, we'll embark on how we are leveraging recent research – and try to contribute ourselves – on various level of our debugging and profiling pipeline.

At each stage, such as bug prevention or tests-recommendations, I'll show which papers inspired us and how we had to tweak the original ideas, so it applies to video-games.

When we talk about games, there are two kind of perceptions, people who think about the first Formula 1 and people who are playing current gen games. So, we are all on the same page, here's a video of the kind of games Ubisoft produces.

Ubisoft produces large open-worlds where they are no limit to the explorations and players are free to do whatever they desire in this simulated environment.

// Video in the blog-post at laforge.ubisoft.com



UBISOFT IN THE WORLD

AAA GAMES	TECH GROUP / LA FORGE
~16k EMPLOYEES	TECH ARCHITECT / ARS
20 + Offices	DEBUG & PROFILE AAA GAMES

Ubisoft produces AAA games and have around 16 thousand employees scattered across 20 offices.

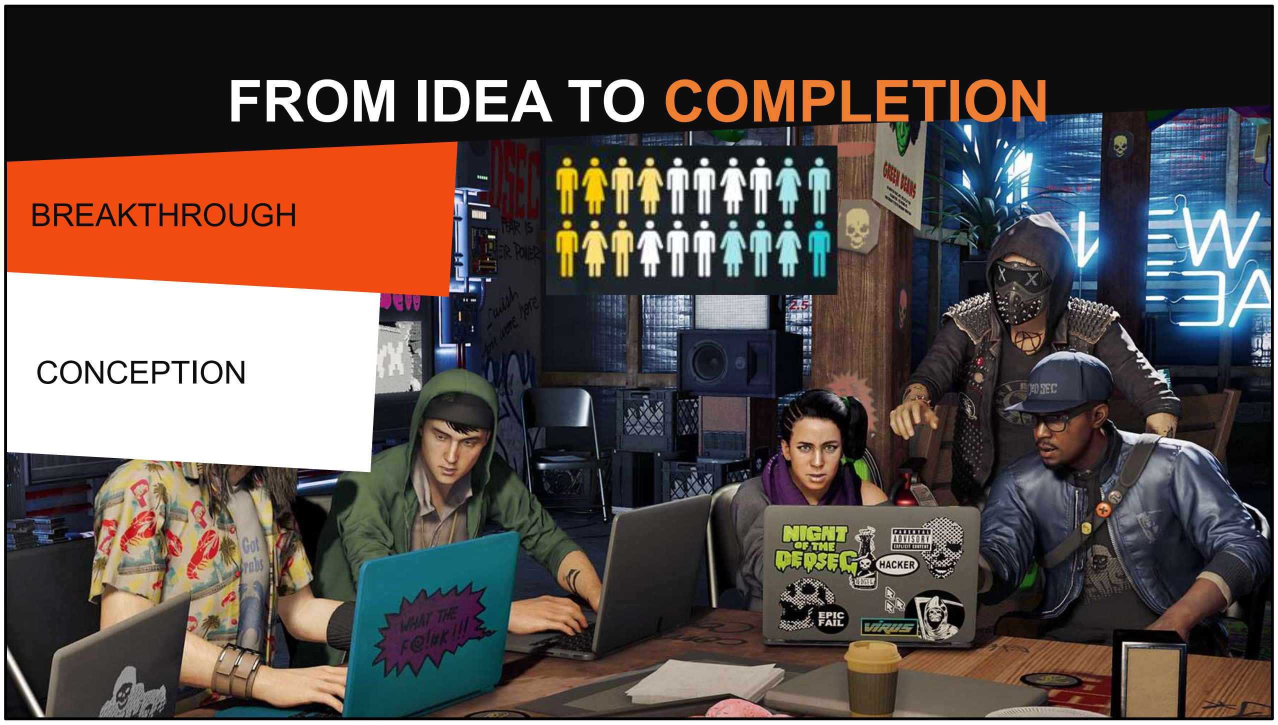
For myself, I wear two hats at Ubisoft. On one hand, I am a technical architect for the technology group where I lead the devs related to debug and profiling at the scale of the compagnie. On the other hand, I am a research scientist for La Forge (Ubisoft Research Lab) where I lead the research roadmap on Software Engineering & Productivity.

MAKING AAA GAMES

FROM IDEA TO COMPLETION

BREAKTHROUGH

From idea to completion, we first begin the creation of a video game by the breakthrough phase where very few people are involved. Usually, all the different skillsets to make a video games are present: creatives, animators, devs, artists and sounds engineers.

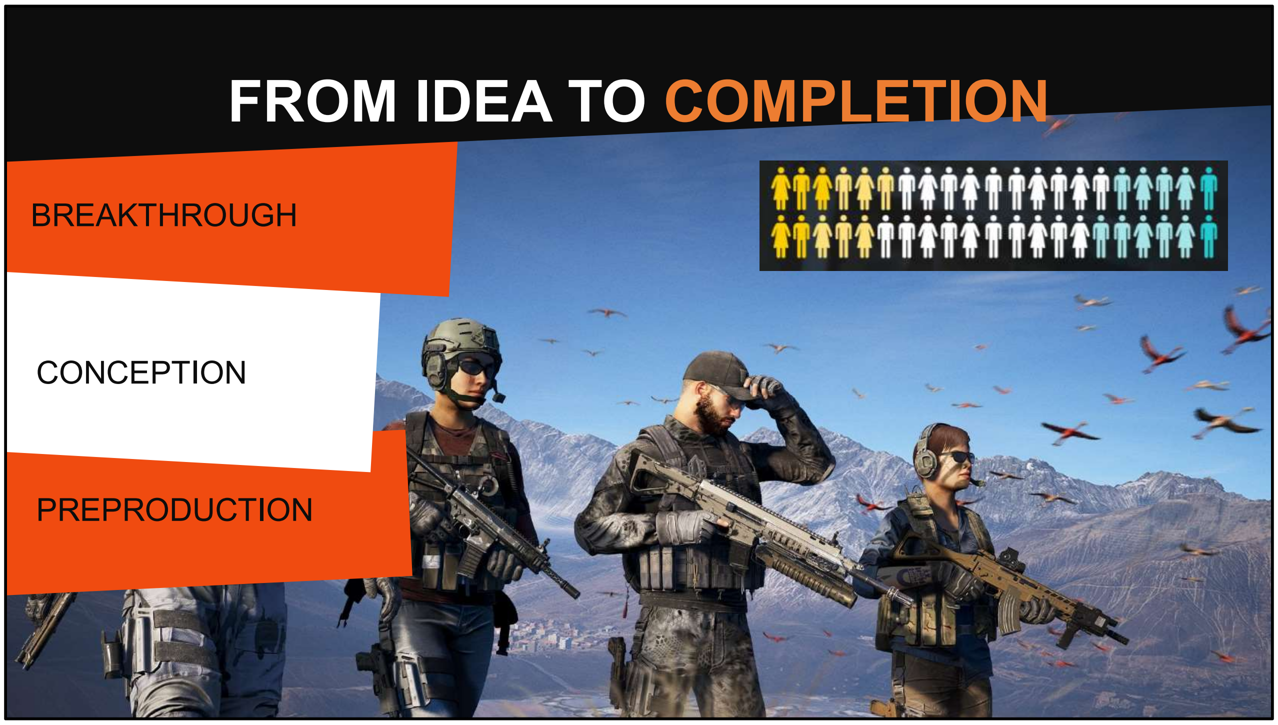


FROM IDEA TO COMPLETION

BREAKTHROUGH

CONCEPTION

Then, we engage the conception phase where we ramp up peoples and skills to produce the game. They must validate that the game is indeed fun to play by exercising some of the game mechanics.



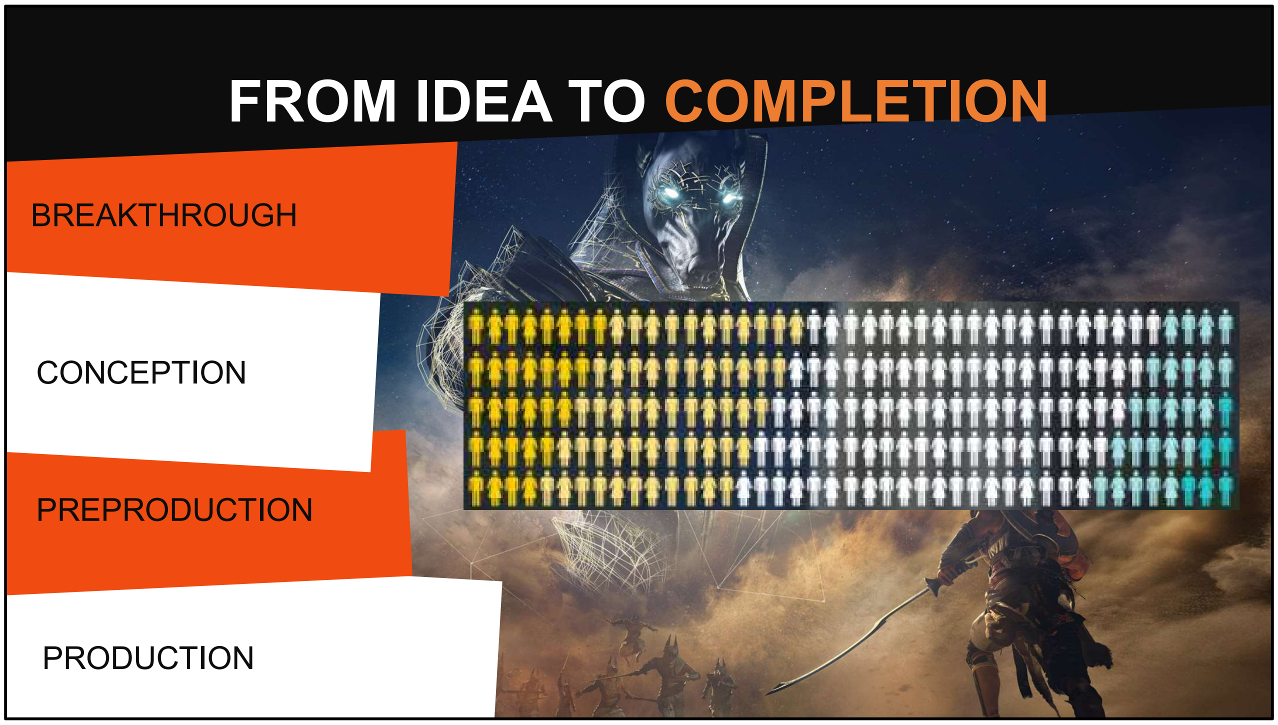
FROM IDEA TO COMPLETION

BREAKTHROUGH

CONCEPTION

PREPRODUCTION

In preproduction, the teams are scale up again in order to push further the exploration of the creative ideas.



FROM IDEA TO COMPLETION

BREAKTHROUGH

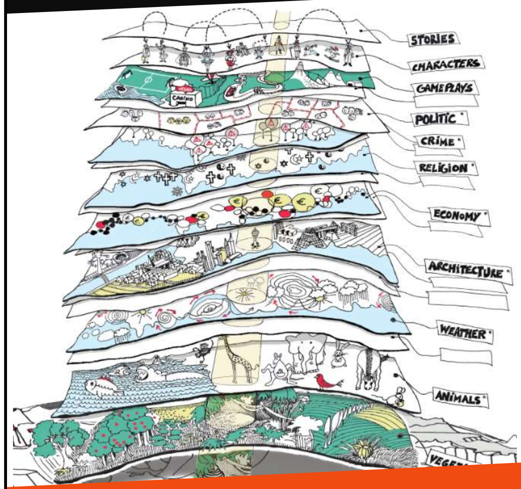
CONCEPTION

PREPRODUCTION

PRODUCTION

Finally, in production, teams are reaching max capacities. Producing a AAA games take hundreds of people.

FROM IDEA TO COMPLETION

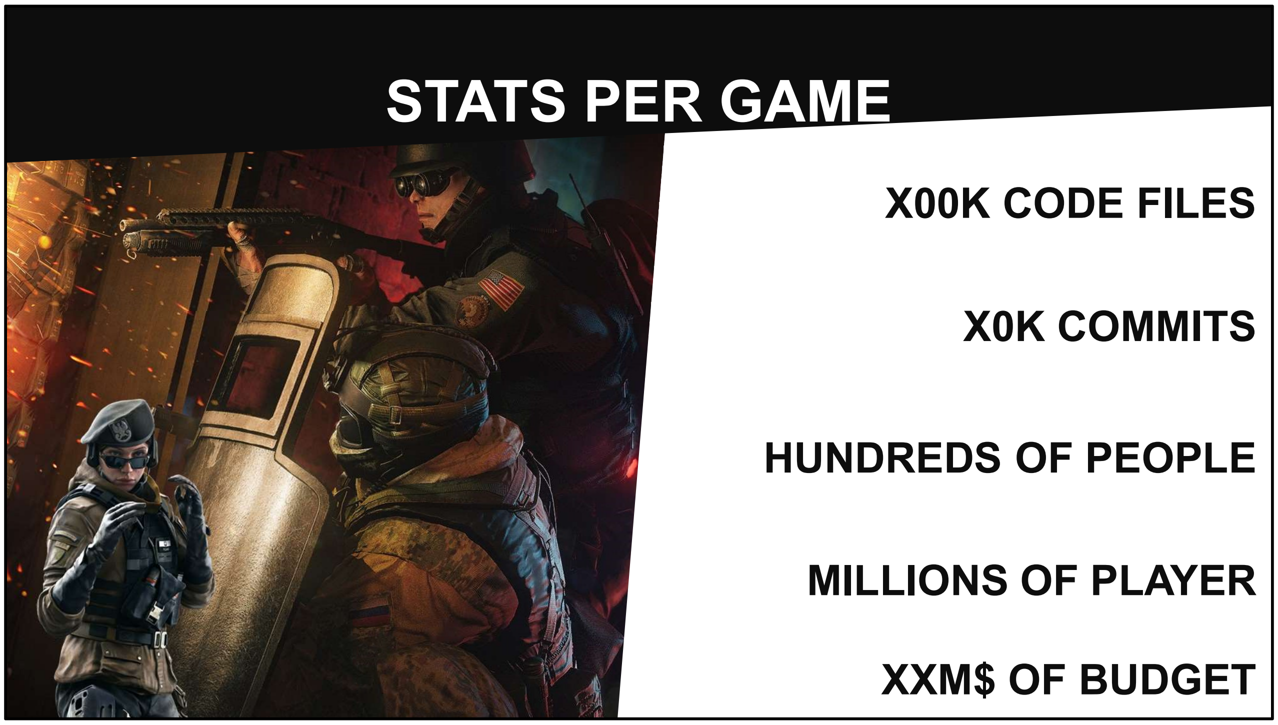


PHYSIQUE OPTIQUE
OPTICS
ANTHROPOLOGIE
ANTHROPOLOGY
BIOMÉCANIQUE
BIOMECHANICS
BOTANIQUE
BOTANY
INTELLIGENCE ARTIFICIELLE
ARTIFICIAL INTELLIGENCE

WORLD SIMULATION

It takes so many people because we are, in the end, building a world simulation, where we have to simulate the vegetation, the animals, the weather, the architecture, the economy, religion, crime, politic, gameplay, characters and then, and top of it all, the stories.

The simulation is not accurate because the game have to be fun to play – if I can only sprint for 15 seconds as I my current shape would allow – it won’t be so fun. So it’s a fun world-simulations. It can, in some context be even more challenging to build than an accurate one as you have to be creative about the laws of nature and physics you want to modify.



STATS PER GAME

X00K CODE FILES

X0K COMMITS

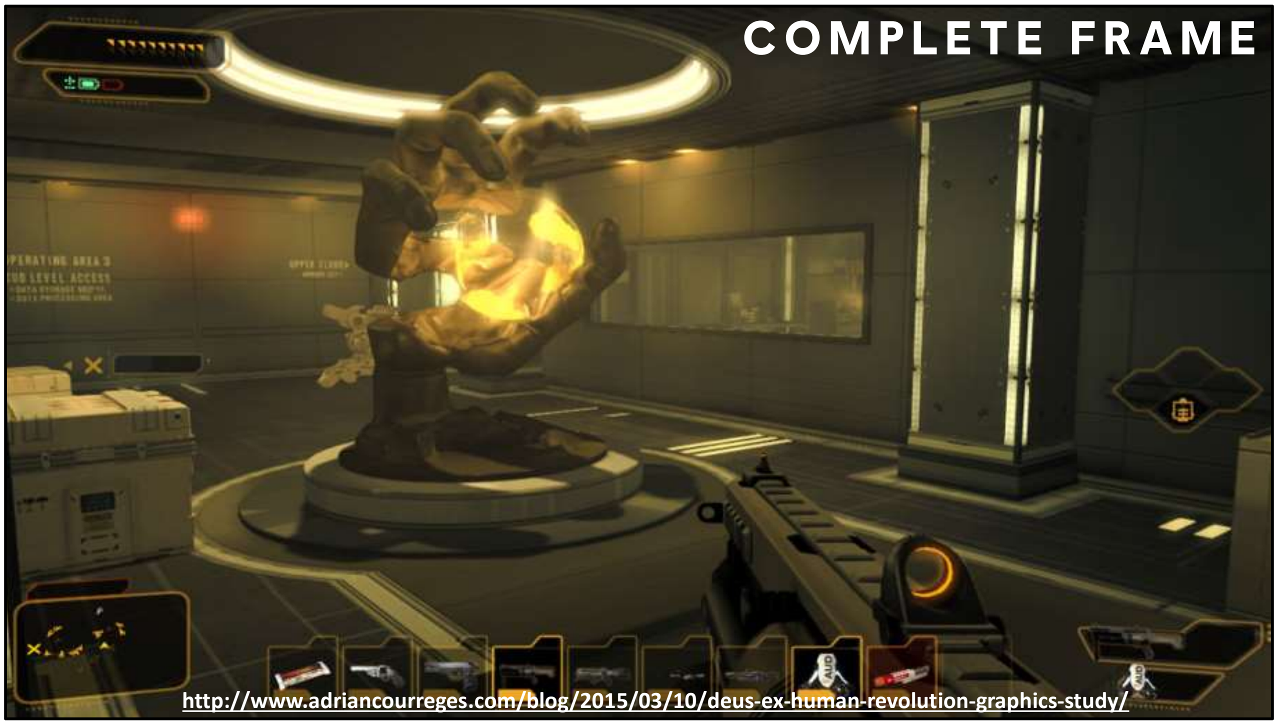
HUNDREDS OF PEOPLE

MILLIONS OF PLAYER

XXM$ OF BUDGET

In the end, a AAA game is hundreds of thousands of files, tens of thousands of commits, hundreds of people and millions of players. It takes tens of millions of dollars to produce. You can think of AAA games as the blockbusters of the video game industry.

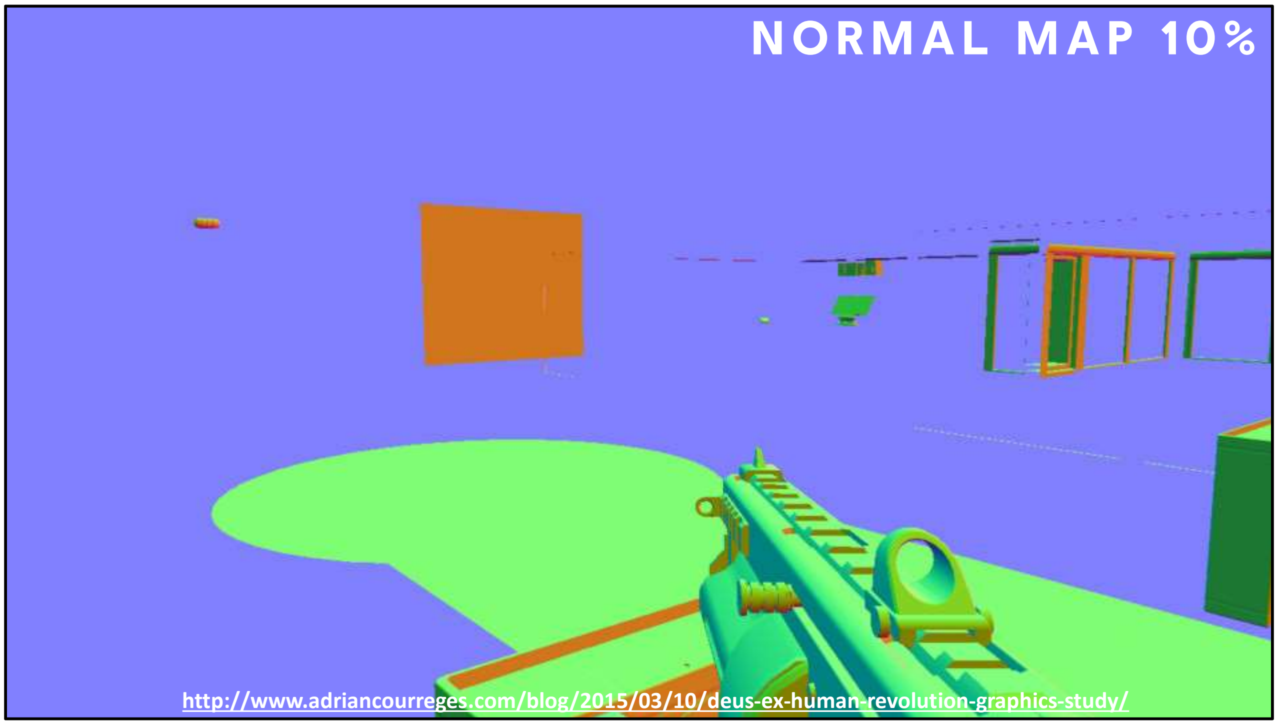
COMPLETE FRAME



http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

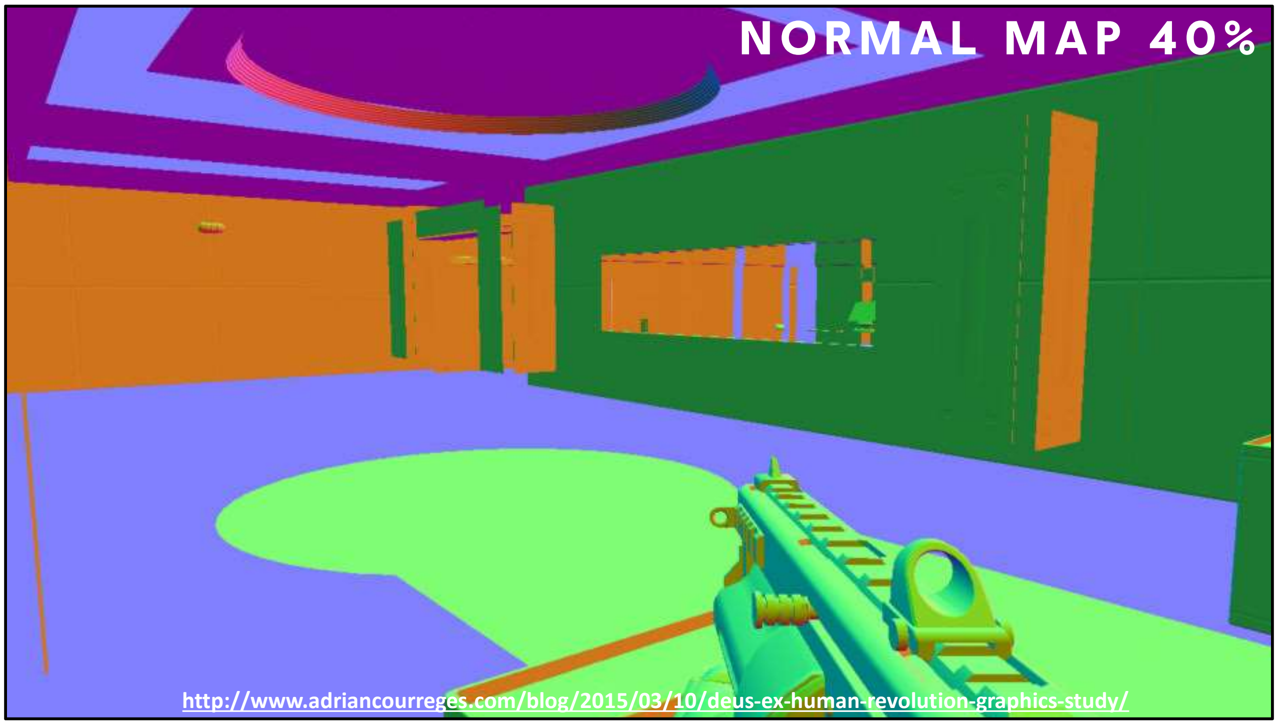
Let’s get a technical about what’s happening on screen when you play. Here’s a complete frame rendered by a game.

NORMAL MAP 10%



http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

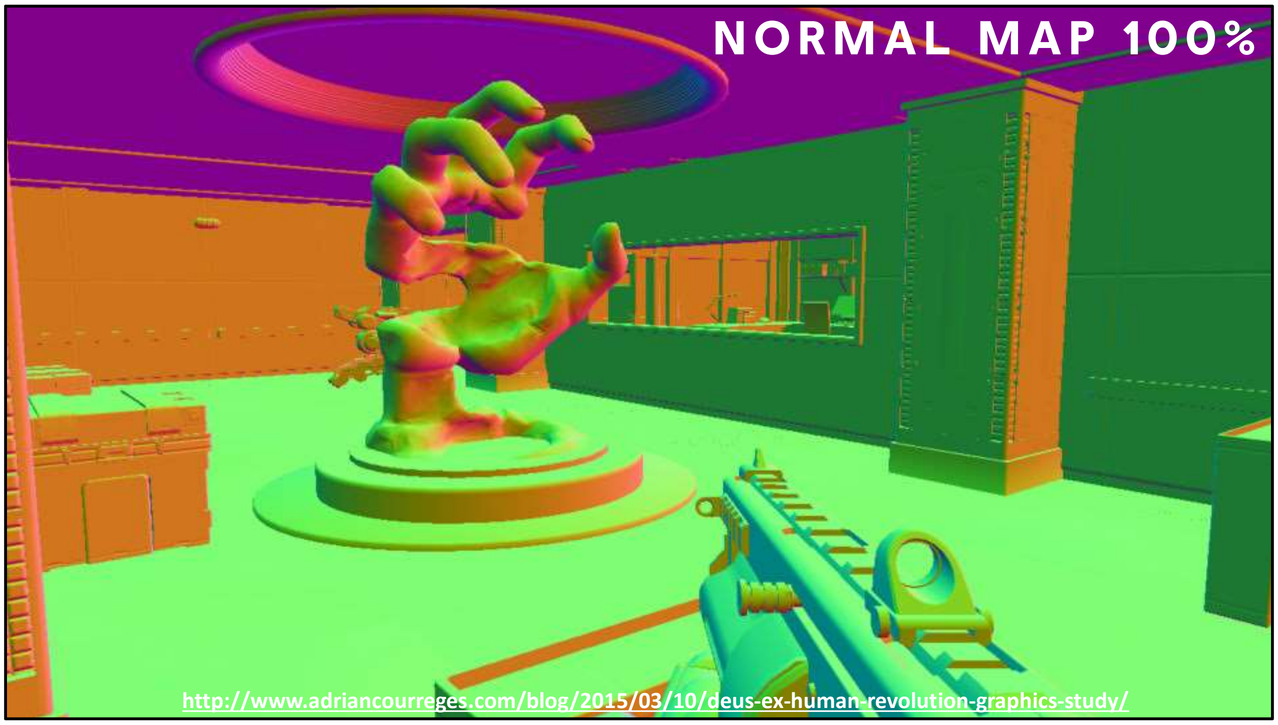
To make it there, there are a lot of steps. First, you load the map. Here’s what it looks like at 10%.



NORMAL MAP 40%

http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

40%

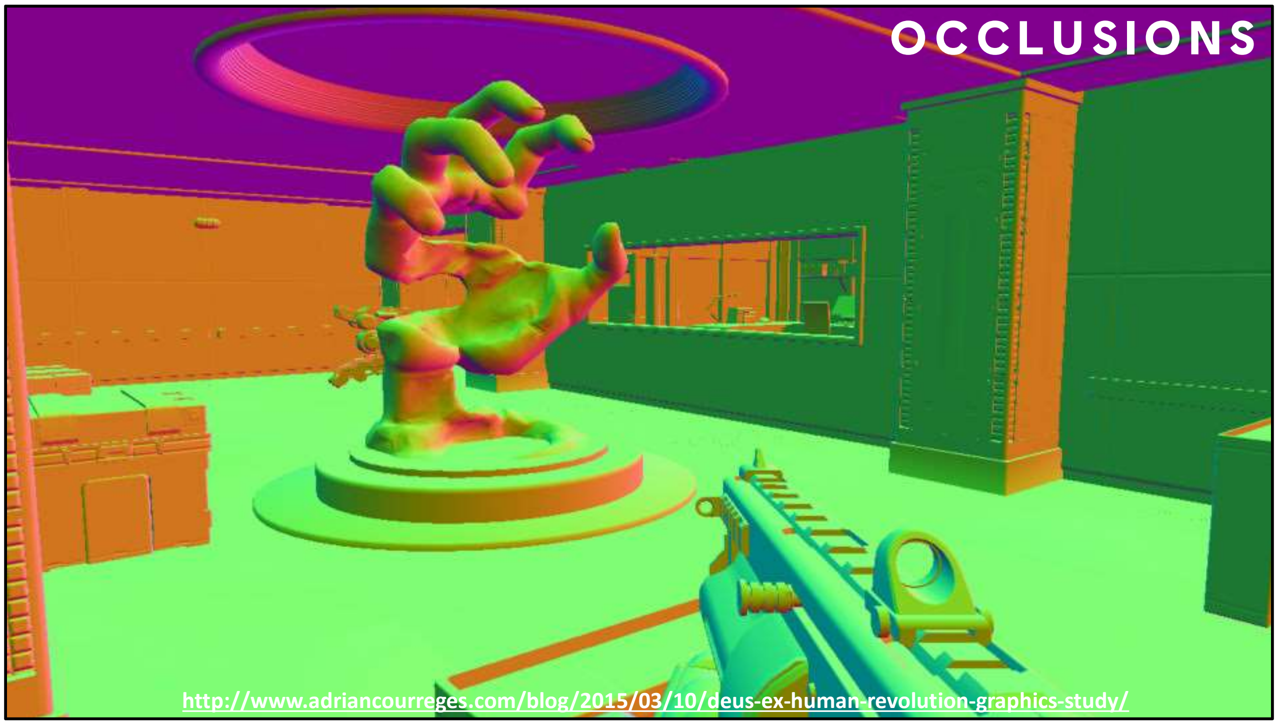

NORMAL MAP 100%
http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

Then 100%

http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

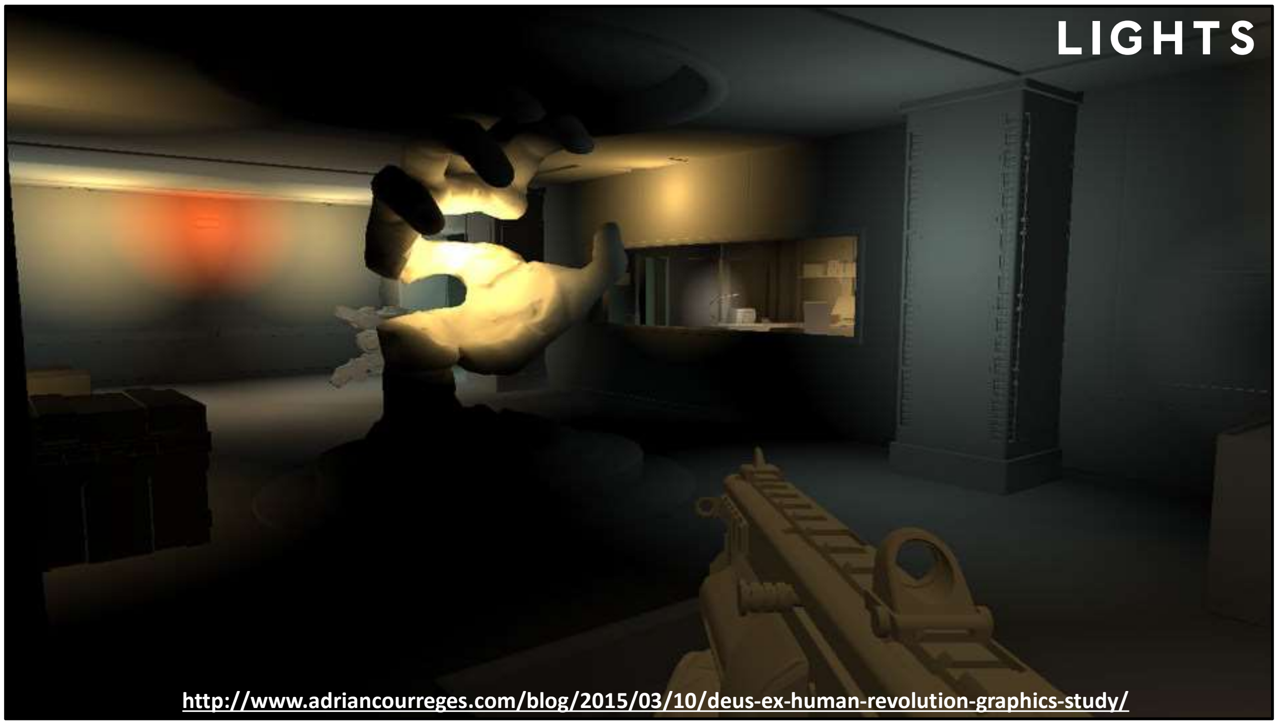
We carry on by loading the shadows,

OCCLUSIONS


http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

Computing the occlusions,

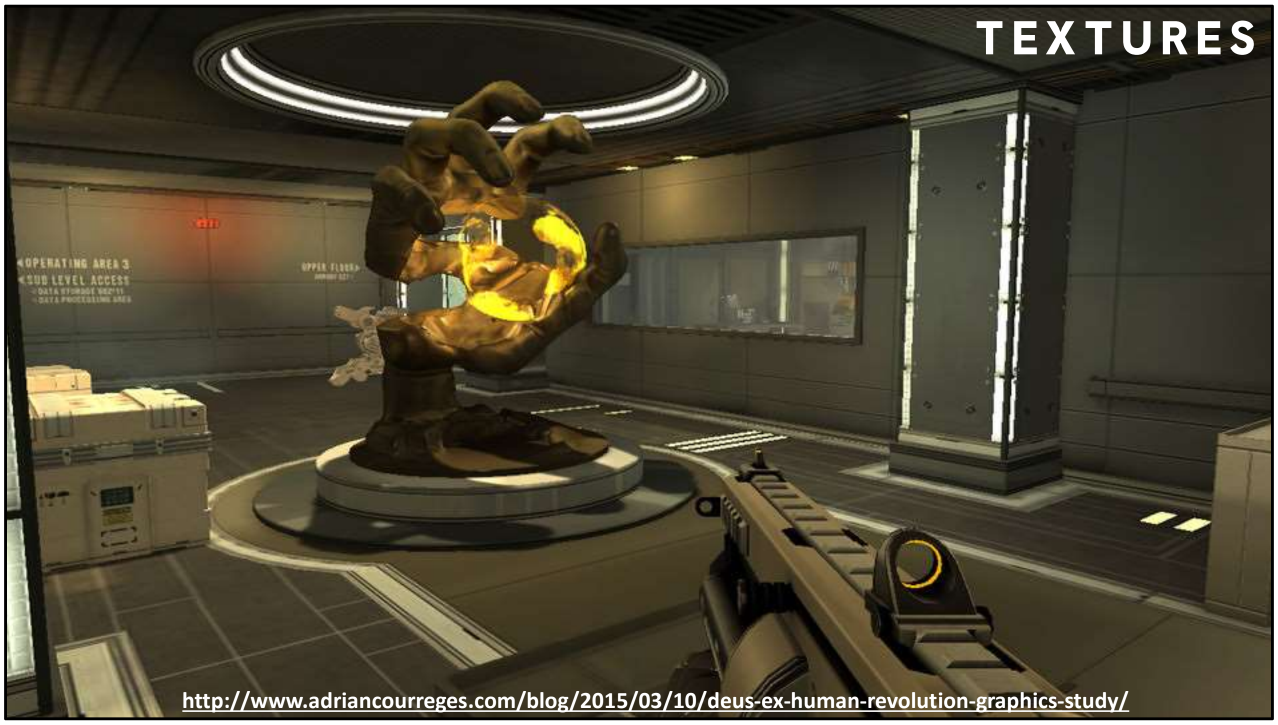
LIGHTS



http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

Lights,

TEXTURES

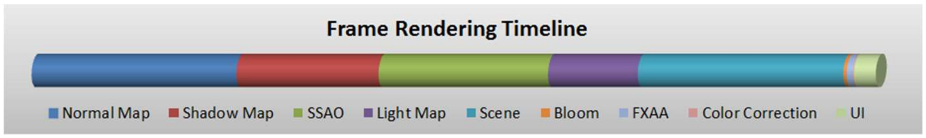


http://www.adriancourreges.com/blog/2015/03/10/deus-ex-human-revolution-graphics-study/

And finally, we load the textures.

This is very expedited and a few dozens additional steps are required in reality.

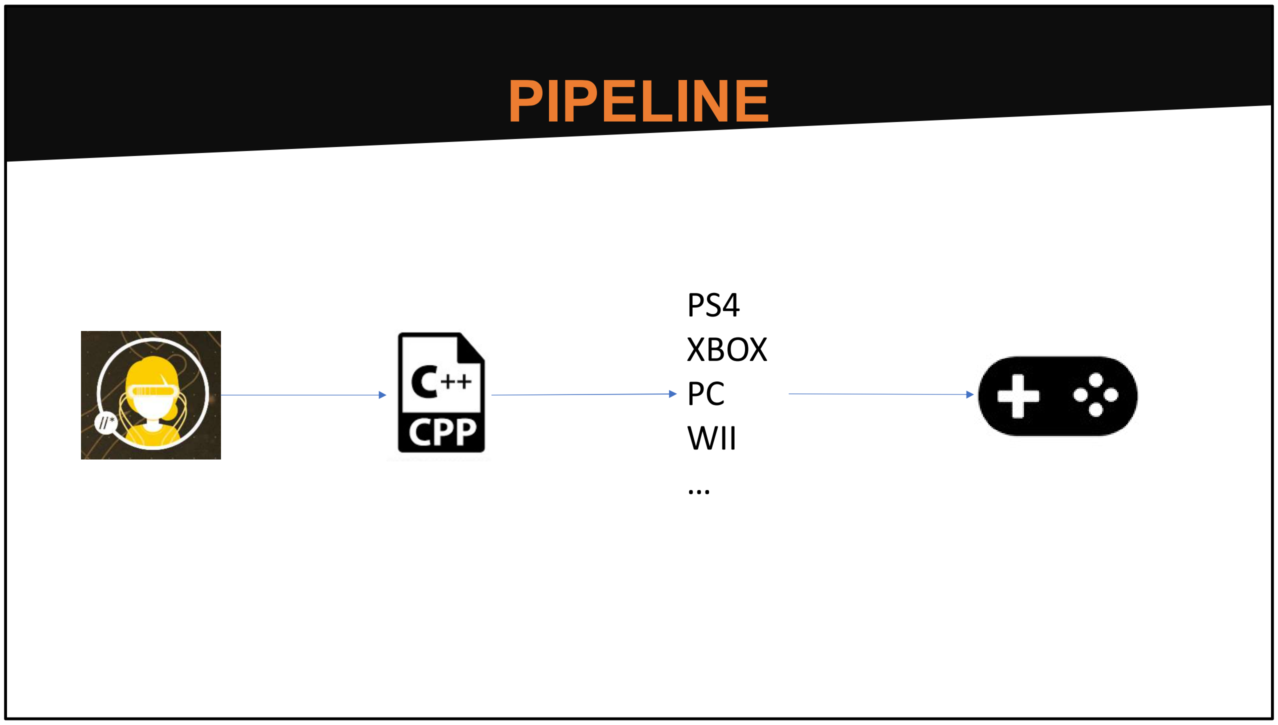
RENDERING



- 60fps ≈ 16 milliseconds per frame
- 30fps ≈ 33 milliseconds per frame
- No gameplay mechanics / physics / online / …

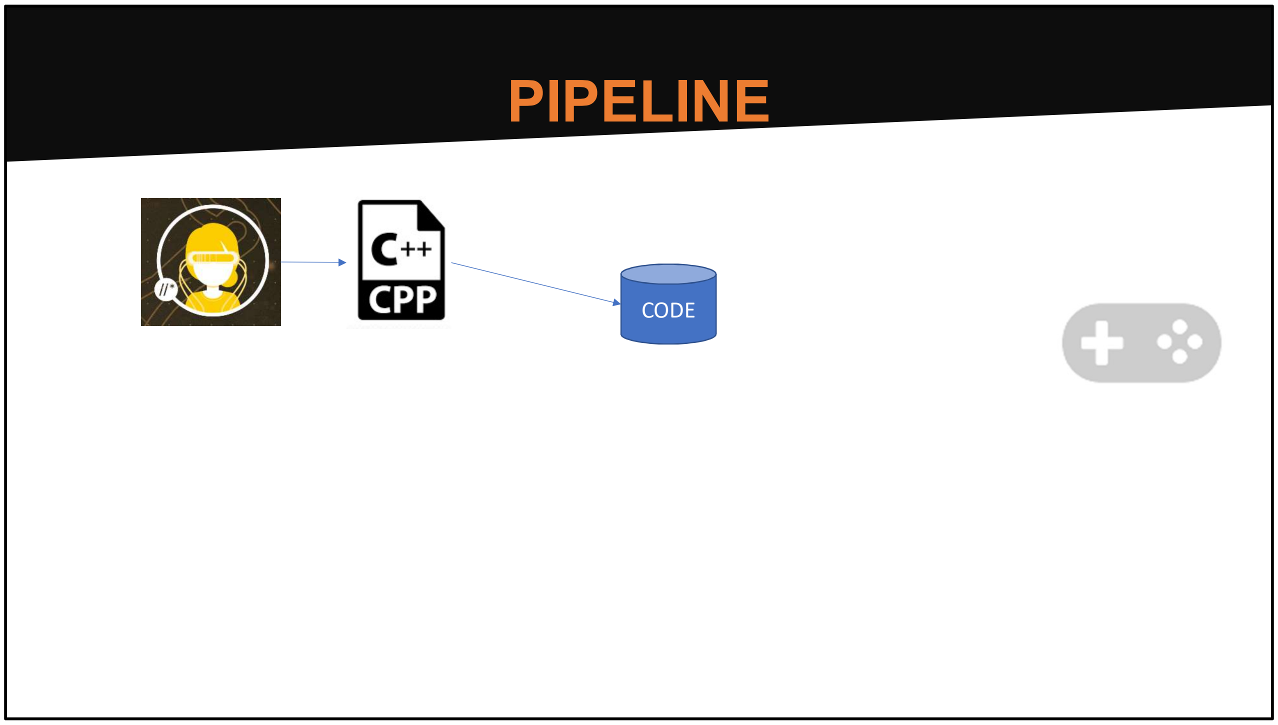
So all of this to render one unique frame. If you are playing on PC, an acceptable frame rate – or the number of frame you want to be displaying per second – is 60 fps. On console it's 30 fps.

It means you have to be able to output a frame every 16 ms on PC and every 33 ms on console. It's already challenging but, in addition of the rendering, you still didn't do any gameplay mechanics / physics computations / online calls and everything that makes the game.



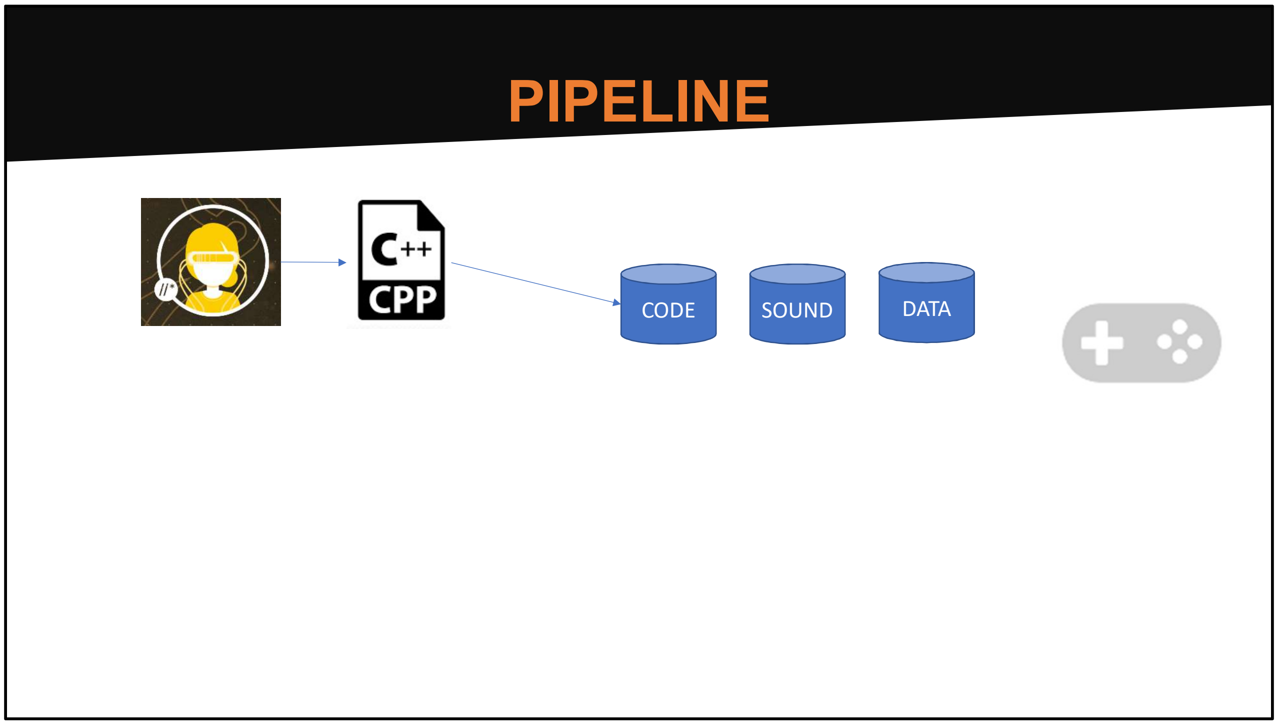
PIPELINE

On the pipeline side of things, we could think that a bunch of devs are writing tons of cpp and we compile it for our different platforms and done, you got a game. It used to be somewhat this way, but with the advent of modern AAA games and games as a service, where the games are receiving a lot of updates after lunch, we had to adapt.

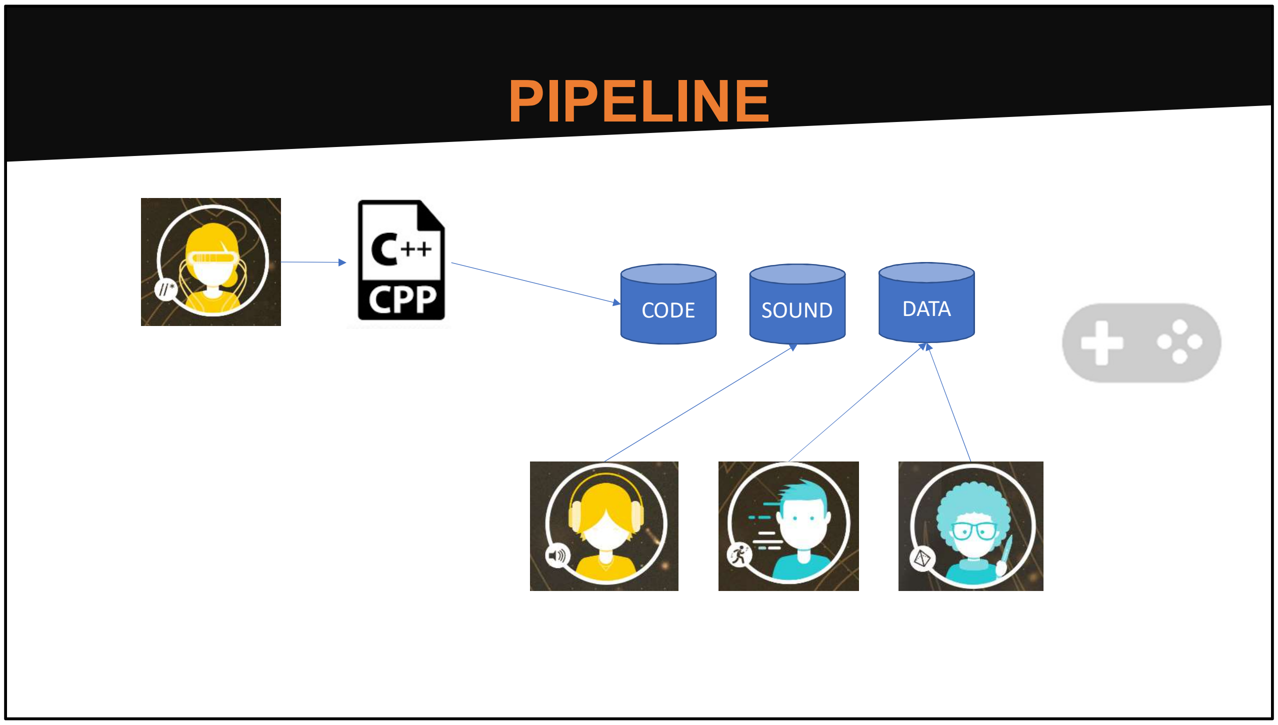


PIPELINE

First, CPP progs are writing cpp that is saved into a perforce repository

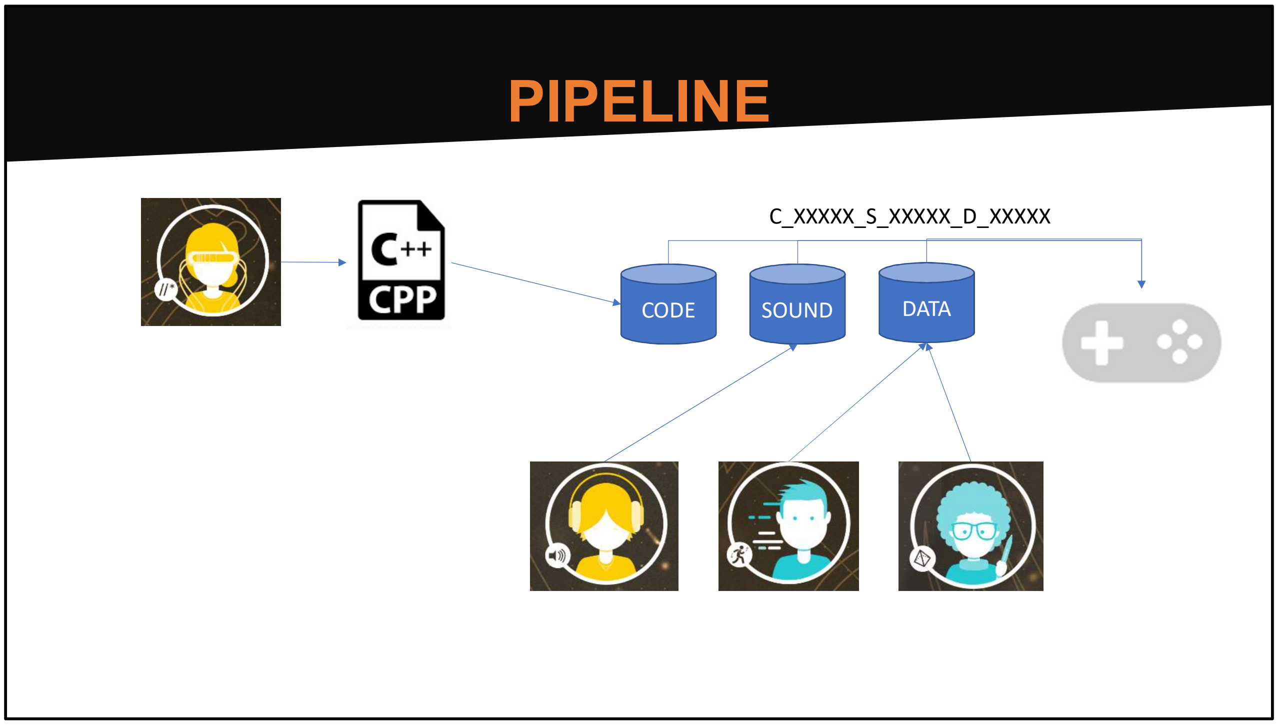


But to make a game, you also need sound and data.



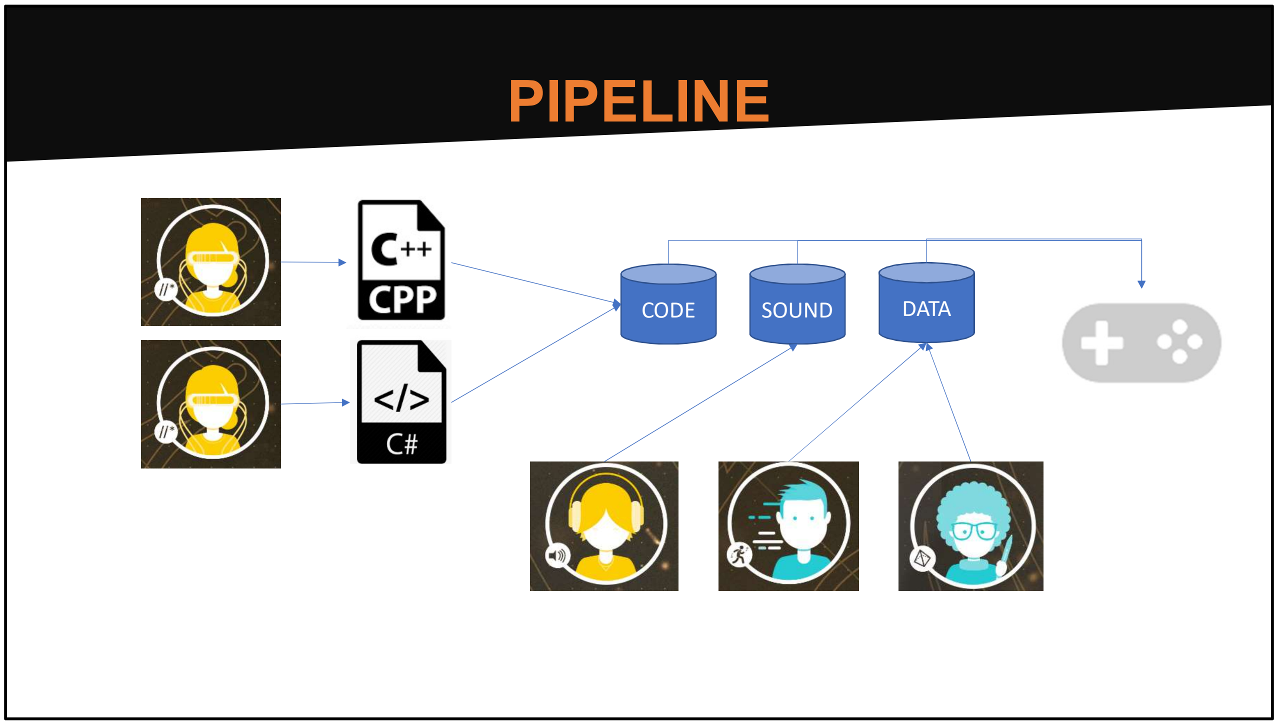
PIPELINE

This sound and data assets are produced by sound engineers, animators and artists.



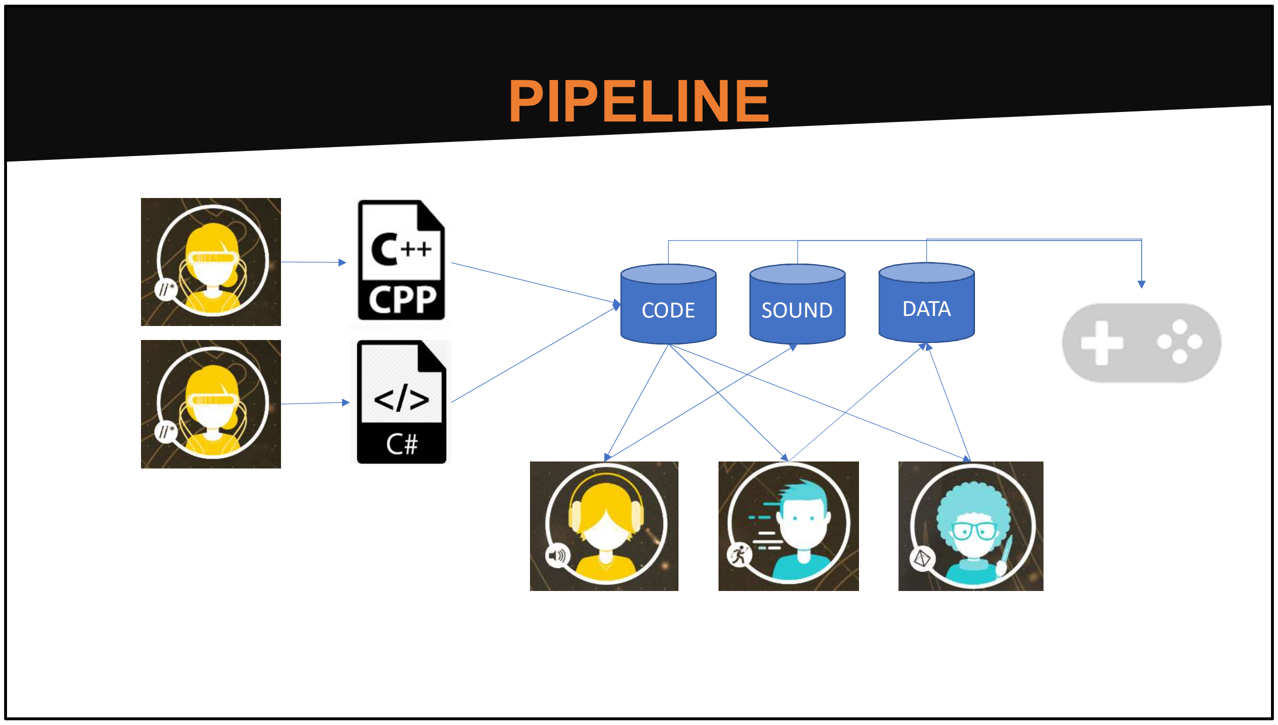
PIPELINE

For us, a game version is not your typical semantic versioning but a combination of versions from code, data, and sound.



PIPELINE

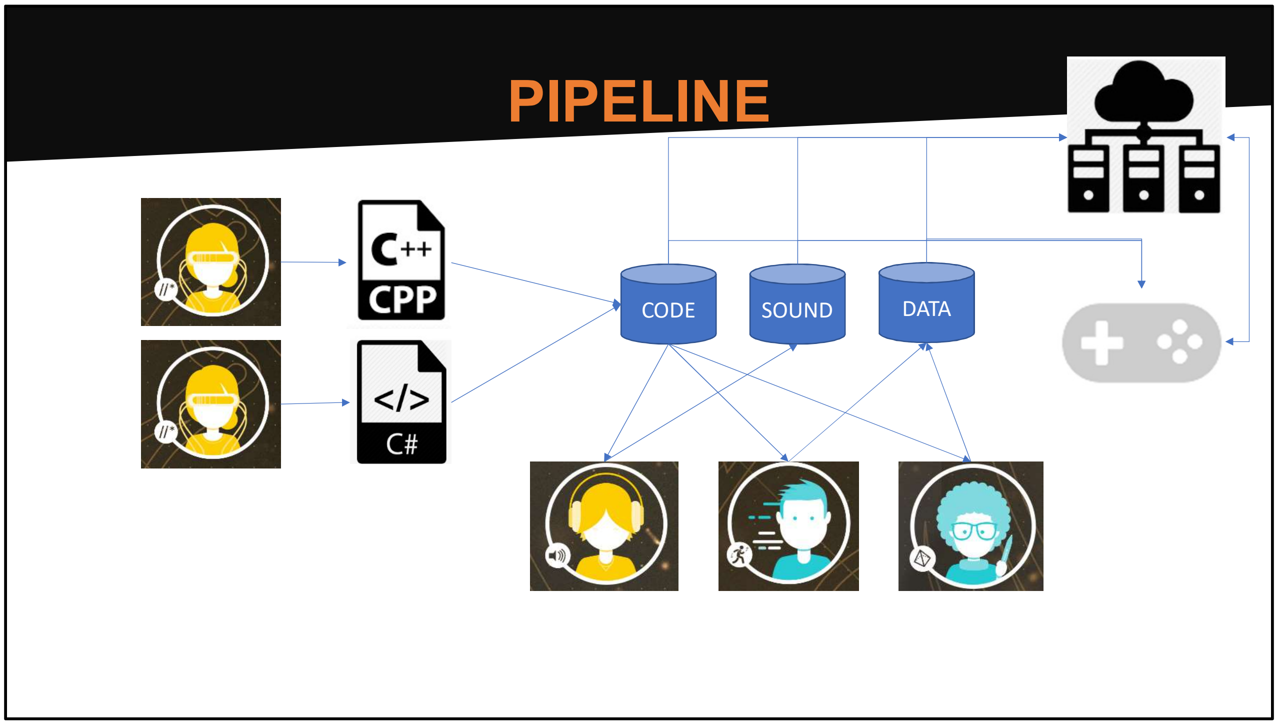
Artists, sound engineers and animators require tools to be able to perform their work. We buy some of them and we make the ones that are custom to our games ourselves. Most of these tools are coded in csharp and versioned in our repositories.



PIPELINE

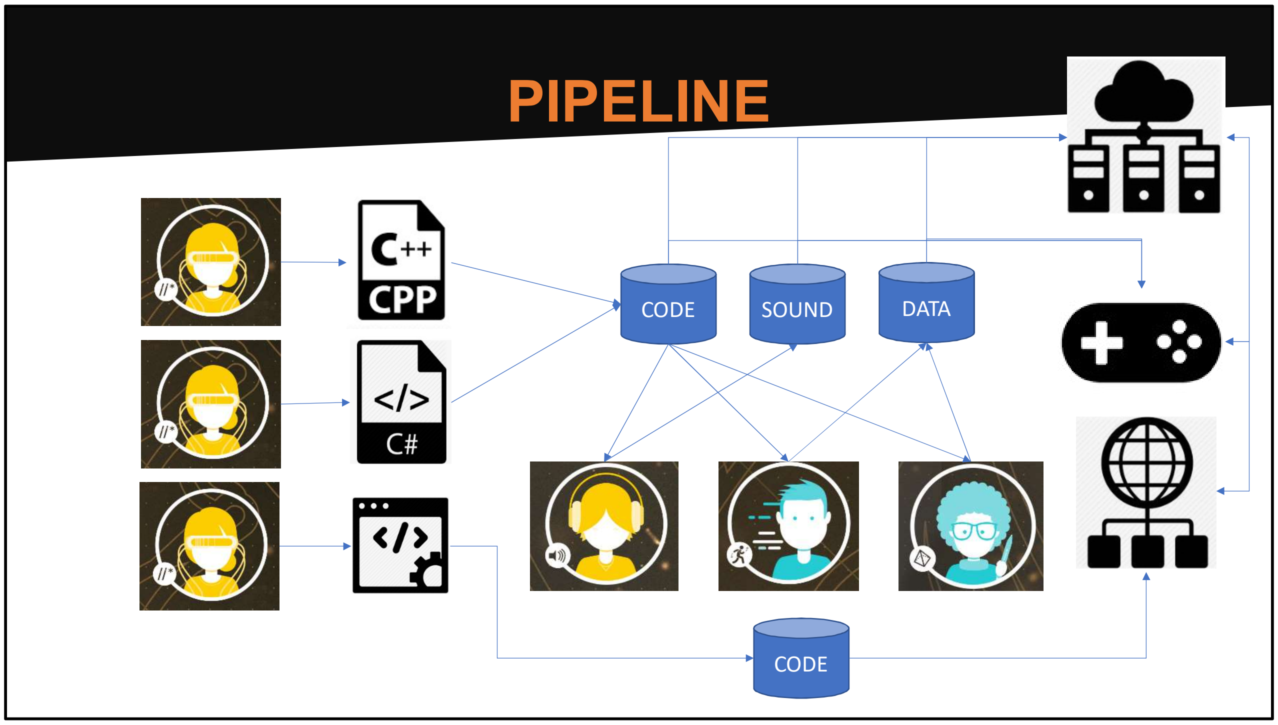
These tools are then distributed to artists, sound engineers and animators

PIPELINE

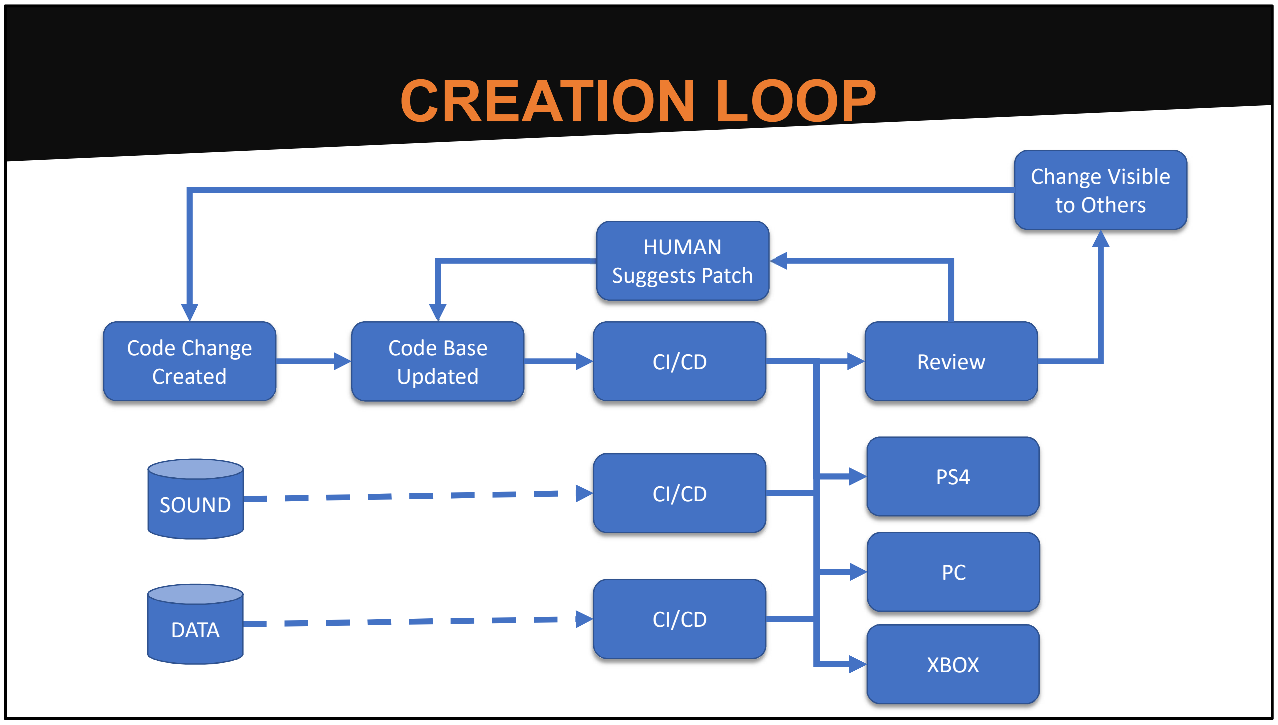


Games now are online and require a server to run. A little known fact is that most of the code between the server and the client is actually the same because you want to make sure you are running the exact same physics engine on both sides.

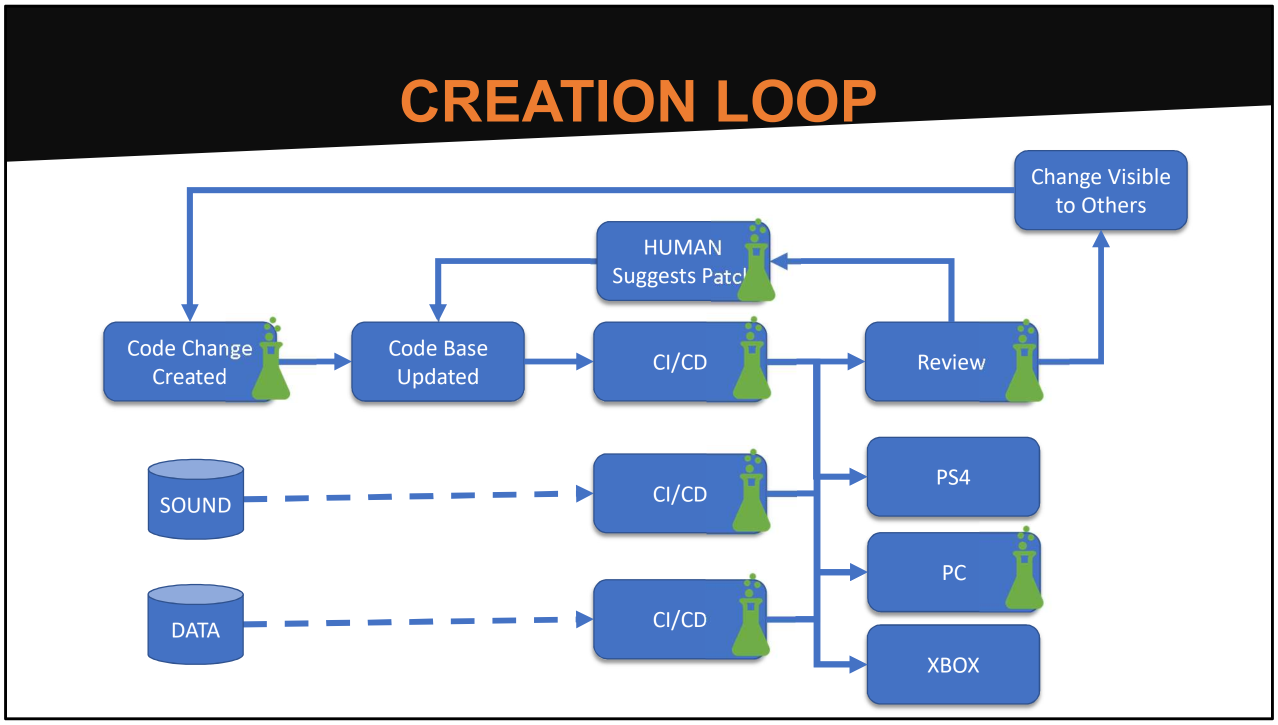
PIPELINE



And … it’s not done yet, games as a service are supported by a lot of extra-functionalities such as friends lists, user generated content, achievements, loots and more that have to be created by yet another breed of programmers. These programmers are online programmers and create web-services that supports the game.

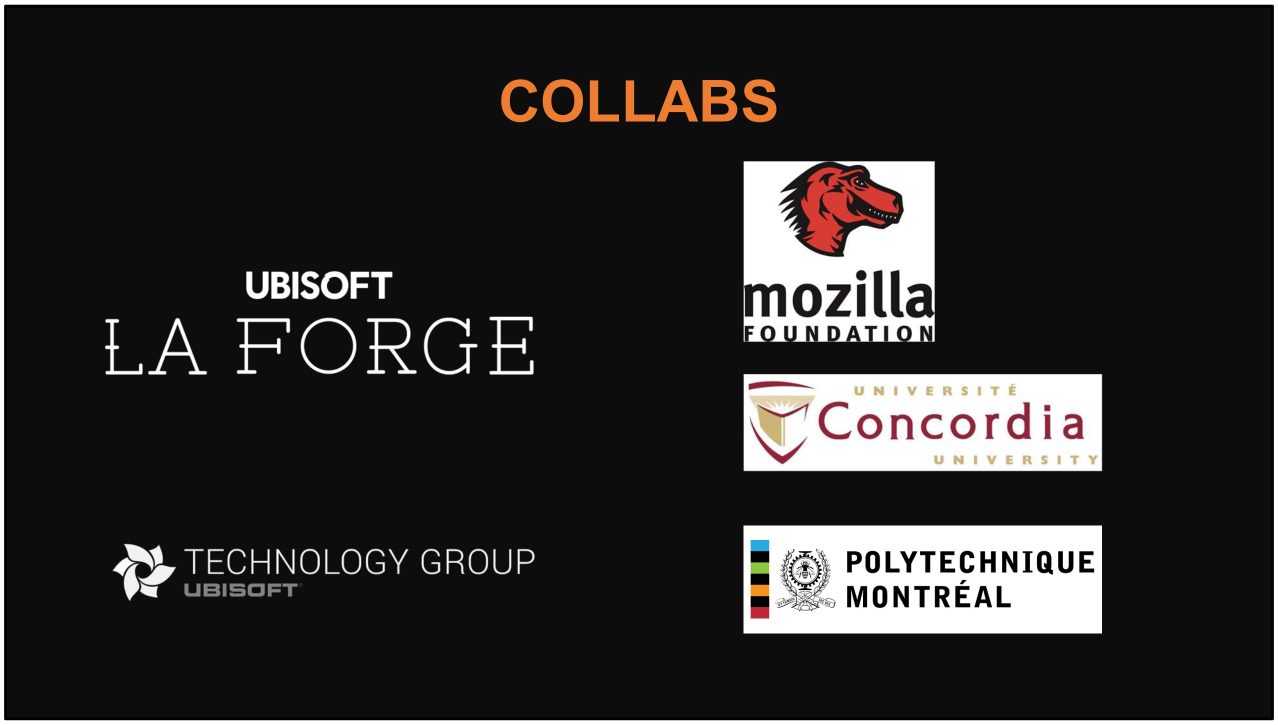


All of this is supported by a classic contribution pipeline.



CREATION LOOP

In which we try to distillate and adapt recent research ideas published in various SE conferences such as ASE, ISCE, ICSME, ICPC, MSR, SANER, …



COLLABS

UBISOFT LA FORGE

TECHNOLOGY GROUP UBISOFT

mozilla FOUNDATION

UNIVERSITÉ Concordia UNIVERSITY

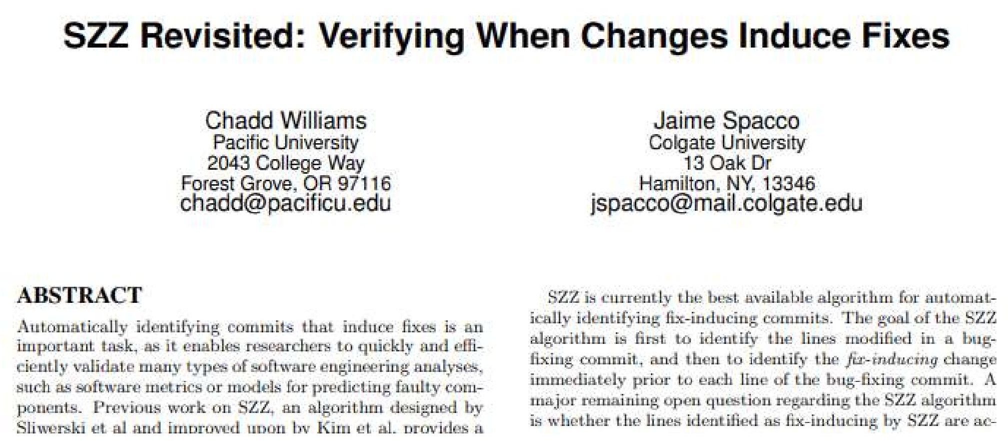
POLYTECHNIQUE MONTRÉAL

Before we go on, I want to emphasize that this is not a one man show and what will be presented today is the work of many teams internally at LaForge and the Technology Group. We are also actively contributing with several universities on open research subject.

Another collaboration we have is with the Mozilla Foundation. With them, we are exchanging code, ideas and skills so we can build a better developer experiences at both companies.

BUG INTRO. PREVENTION

DEBUG



SZZ Revisited: Verifying When Changes Induce Fixes

Chadd Williams
Pacific University
2043 College Way
Forest Grove, OR 97116
chadd@pacificu.edu

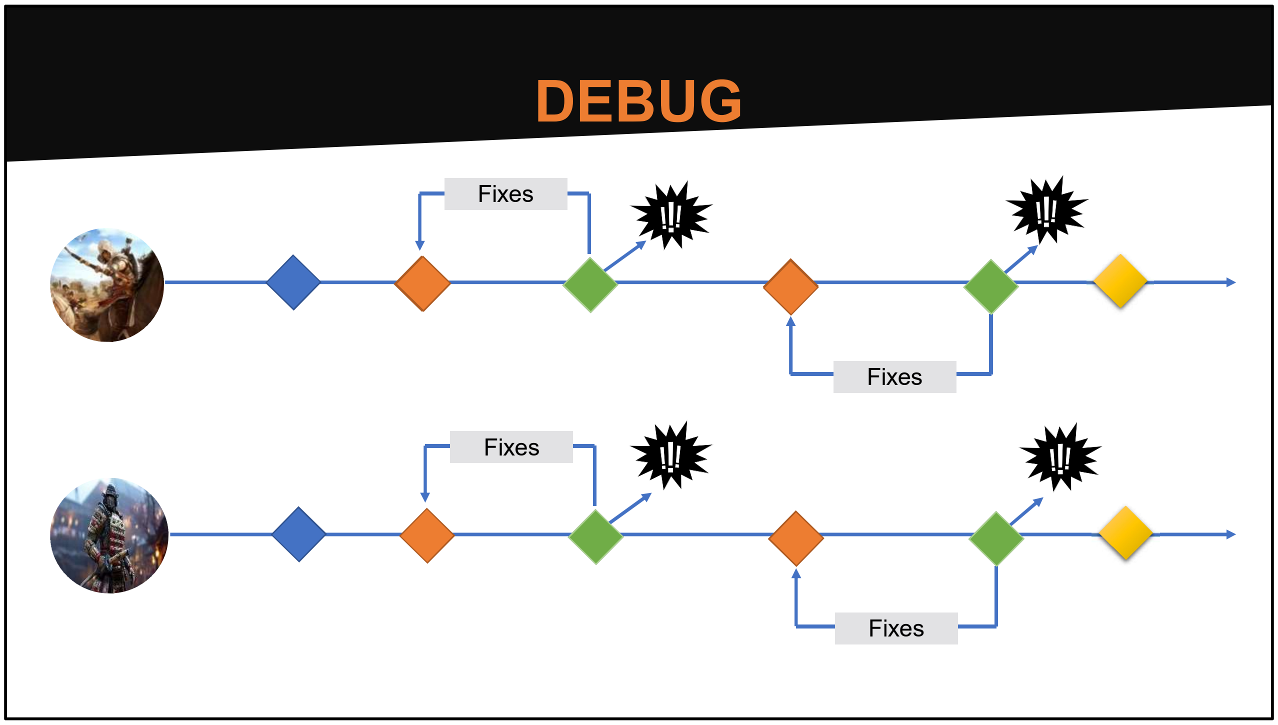
Jaime Spacco
Colgate University
13 Oak Dr
Hamilton, NY, 13346
jspacco@mail.colgate.edu

ABSTRACT

Automatically identifying commits that induce fixes is an important task, as it enables researchers to quickly and efficiently validate many types of software engineering analyses, such as software metrics or models for predicting faulty components. Previous work on SZZ, an algorithm designed by Sliwerski et al and improved upon by Kim et al. provides a

SZZ is currently the best available algorithm for automatically identifying fix-inducing commits. The goal of the SZZ algorithm is first to identify the lines modified in a bug-fixing commit, and then to identify the *fix-inducing* change immediately prior to each line of the bug-fixing commit. A major remaining open question regarding the SZZ algorithm is whether the lines identified as fix-inducing by SZZ are ac-

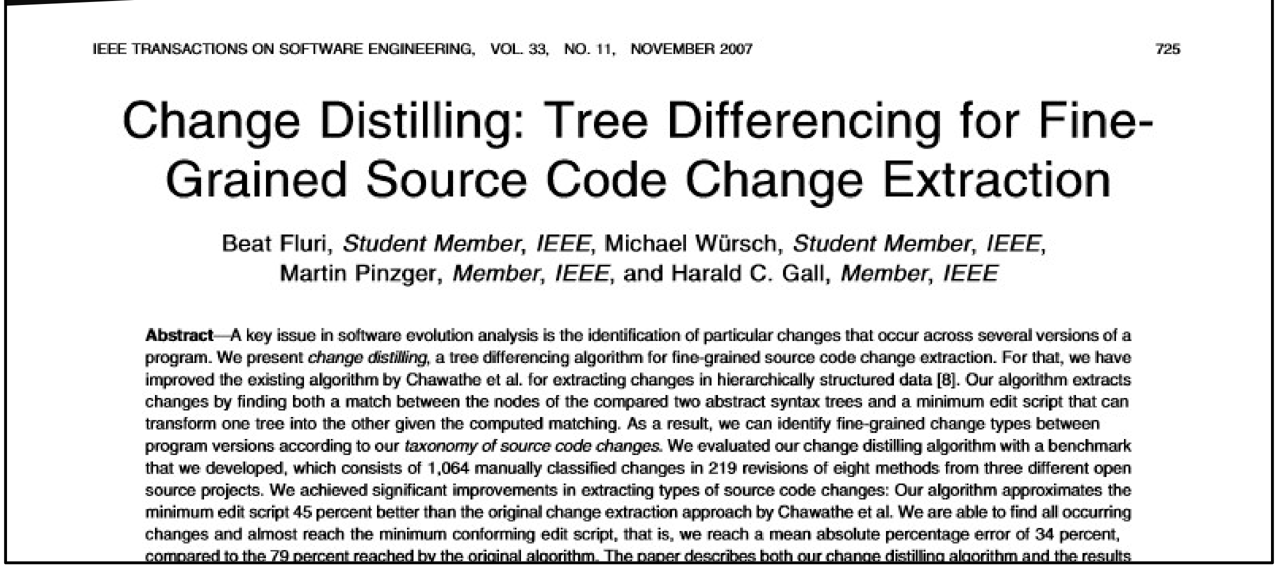
A lot of talks during this conference have explained before what SZZ is and how it works, let's dive into it for the last time this week…



During the software creation process, commits are created, bugs are found and fixed. When a bug is fixed, usually, the developer usually describes it in the commit-message. Using NLP, we can then categorize the commit as a fix-commit and perform a blame operation on it to discover the bug introducing commit.

One of the first modification we've done to the algorithm is that we do not rely on NLP to say if a given commit is a fix-commit. See, we are using JIRA as a ticket management system and it is mandatory and enforced to link your commits to the bugs you are fixing. There's no *guessing* we know what the type of the modification is: maintenance, preventive maintenance, features, chore and so on. Of course, the JIRAs could be misclassified, and they sometime are, but overall, we achieve a higher level of confidence this way.

DEBUG

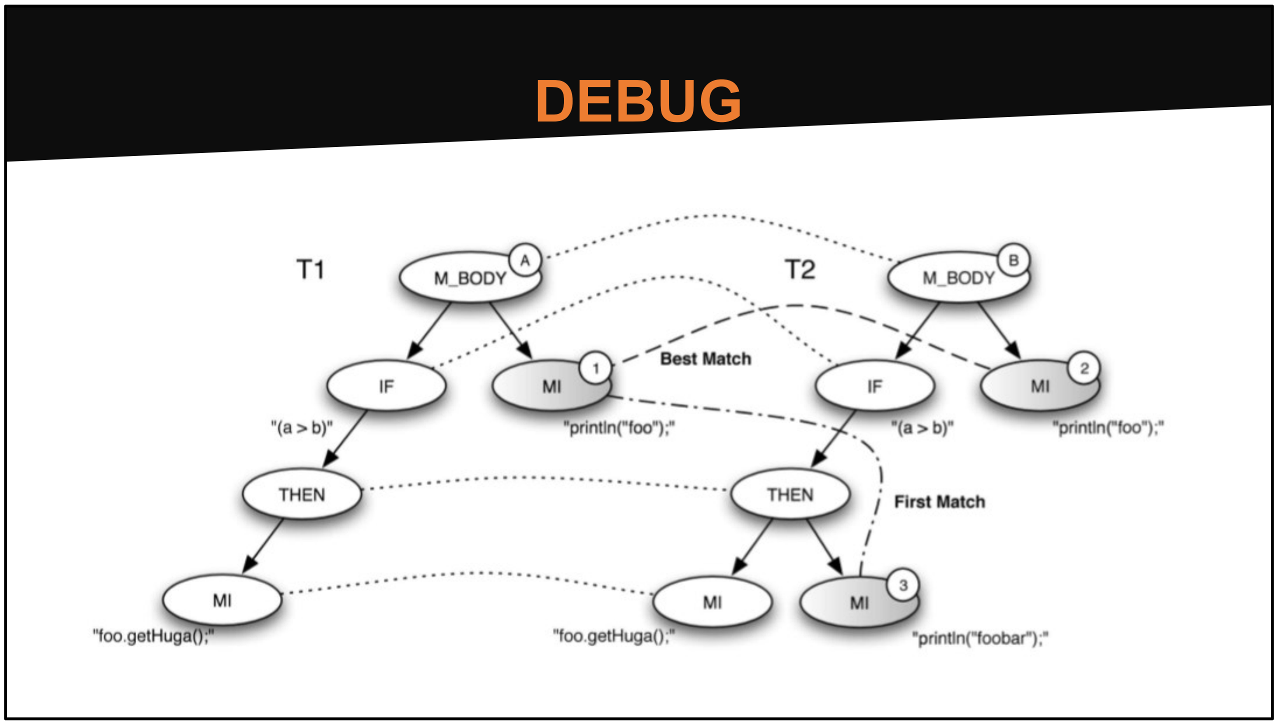

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 11, NOVEMBER 2007 725

Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction

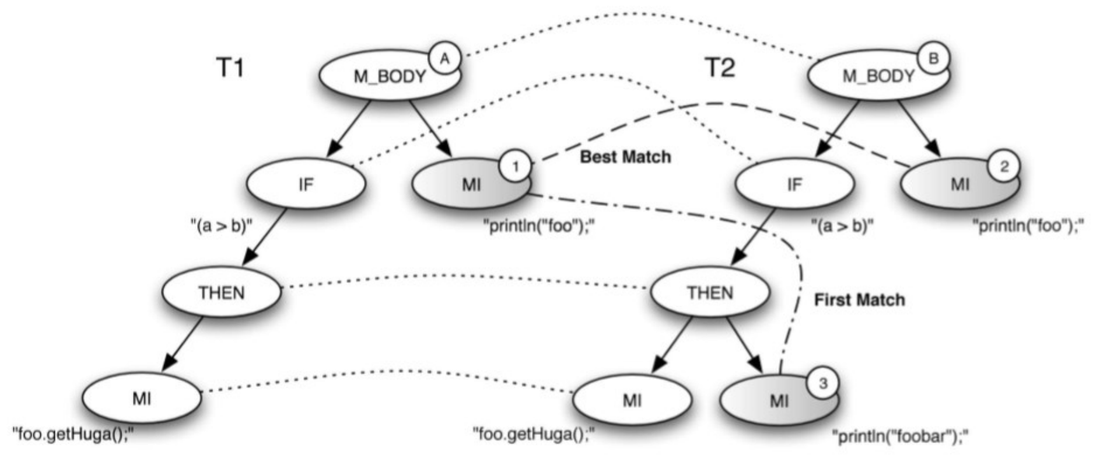
Beat Fluri, *Student Member, IEEE*, Michael Würsch, *Student Member, IEEE*, Martin Pinzger, *Member, IEEE*, and Harald C. Gall, *Member, IEEE*

Abstract—A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. We present *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. For that, we have improved the existing algorithm by Chawathe et al. for extracting changes in hierarchically structured data [8]. Our algorithm extracts changes by finding both a match between the nodes of the compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. As a result, we can identify fine-grained change types between program versions according to our *taxonomy of source code changes*. We evaluated our change distilling algorithm with a benchmark that we developed, which consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. We achieved significant improvements in extracting types of source code changes: Our algorithm approximates the minimum edit script 45 percent better than the original change extraction approach by Chawathe et al. We are able to find all occurring changes and almost reach the minimum conforming edit script, that is, we reach a mean absolute percentage error of 34 percent, compared to the 79 percent reached by the original algorithm. The paper describes both our change distilling algorithm and the results

Another paper that influences us a lot is this one by Fluri et al published more than 12 years ago in TSE. This paper describes how to extract fine-grained changes from two ASTs.

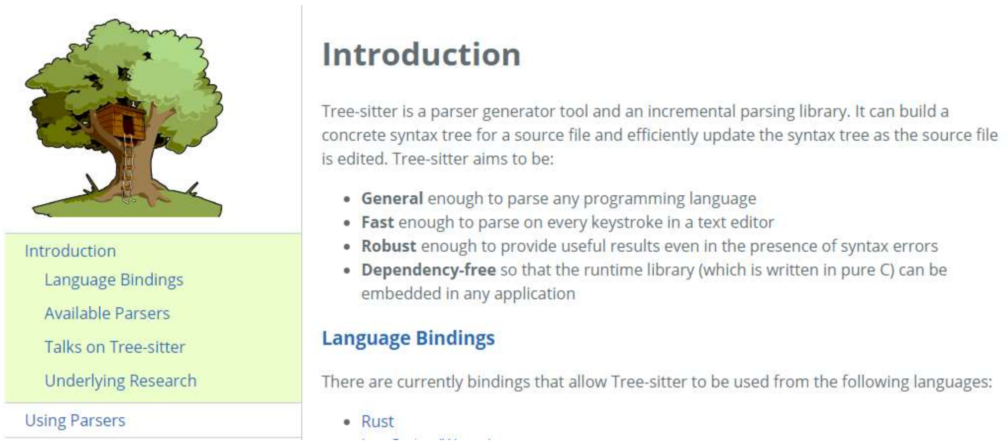


DEBUG



The way it works is that is by comparing AST node by their values, of course, but also the similarity of their subtrees. We use this at a lot of places in our pipelines but, one of them is right here when we try to understand bug introducing changes. We analyze the changes to be able to say, we a higher degree of certainty than using git blame only, that a commit did introduced a defect.

DEBUG



Introduction

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited. Tree-sitter aims to be:

- **General** enough to parse any programming language
- **Fast** enough to parse on every keystroke in a text editor
- **Robust** enough to provide useful results even in the presence of syntax errors
- **Dependency-free** so that the runtime library (which is written in pure C) can be embedded in any application

Language Bindings

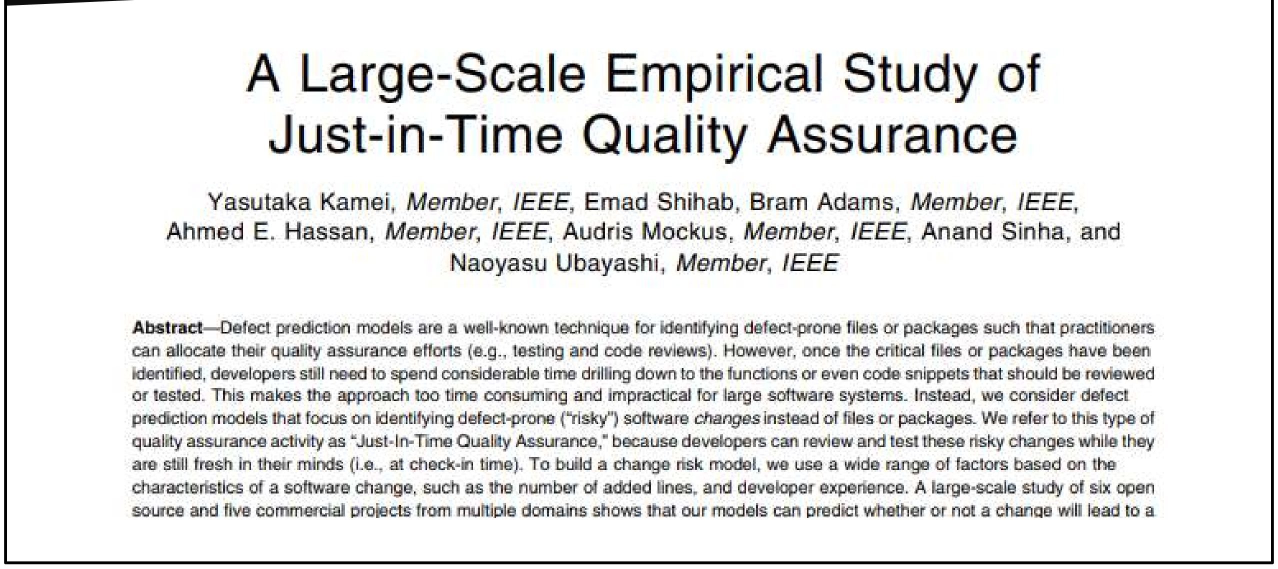
There are currently bindings that allow Tree-sitter to be used from the following languages:

- Rust

http://tree-sitter.github.io/tree-sitter/

To extract ASTs from code-changes we use tree-sitter, an open-source library, that handles a lot of languages including cpp, csharp, typescript, python and more. We did a few contributions to it and our collaborators at Mozilla significantly enhanced the cpp support.

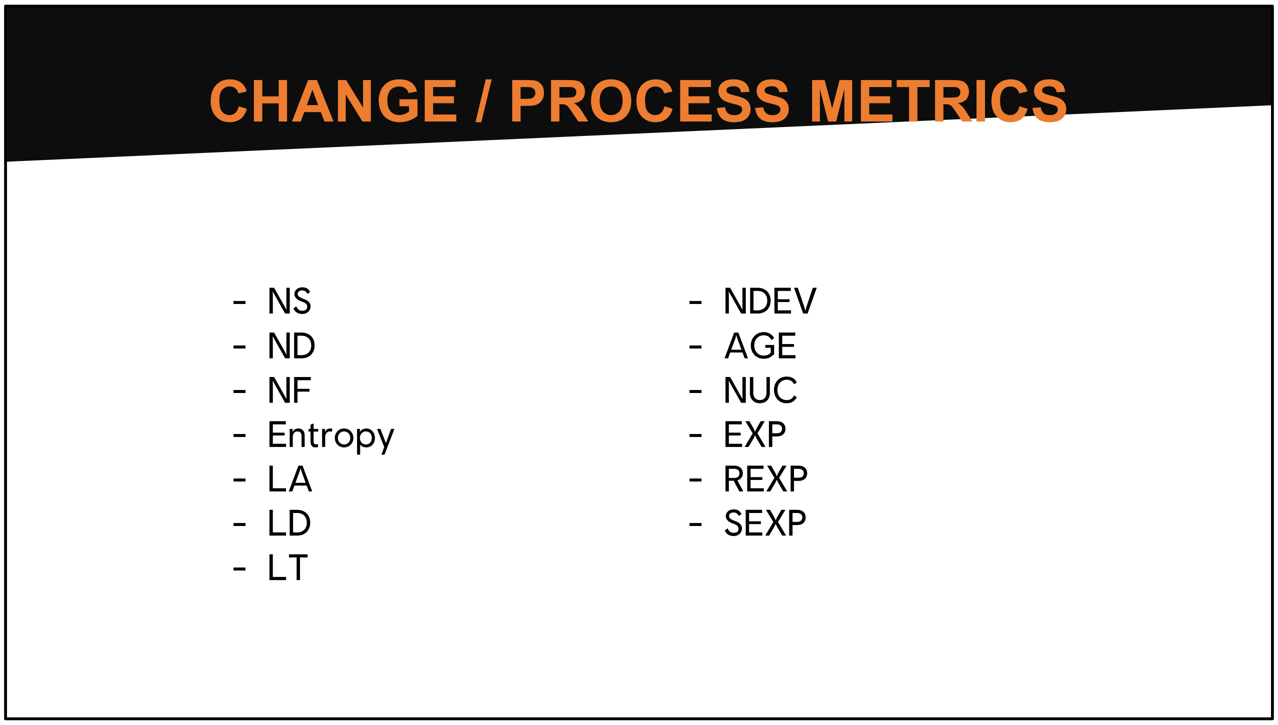
DEBUG


A Large-Scale Empirical Study of Just-in-Time Quality Assurance

Yasutaka Kamei, *Member, IEEE*, Emad Shihab, Bram Adams, *Member, IEEE*, Ahmed E. Hassan, *Member, IEEE*, Audris Mockus, *Member, IEEE*, Anand Sinha, and Naoyasu Ubayashi, *Member, IEEE*

Abstract—Defect prediction models are a well-known technique for identifying defect-prone files or packages such that practitioners can allocate their quality assurance efforts (e.g., testing and code reviews). However, once the critical files or packages have been identified, developers still need to spend considerable time drilling down to the functions or even code snippets that should be reviewed or tested. This makes the approach too time consuming and impractical for large software systems. Instead, we consider defect prediction models that focus on identifying defect-prone ("risky") software *changes* instead of files or packages. We refer to this type of quality assurance activity as "Just-In-Time Quality Assurance," because developers can review and test these risky changes while they are still fresh in their minds (i.e., at check-in time). To build a change risk model, we use a wide range of factors based on the characteristics of a software change, such as the number of added lines, and developer experience. A large-scale study of six open source and five commercial projects from multiple domains shows that our models can predict whether or not a change will lead to a

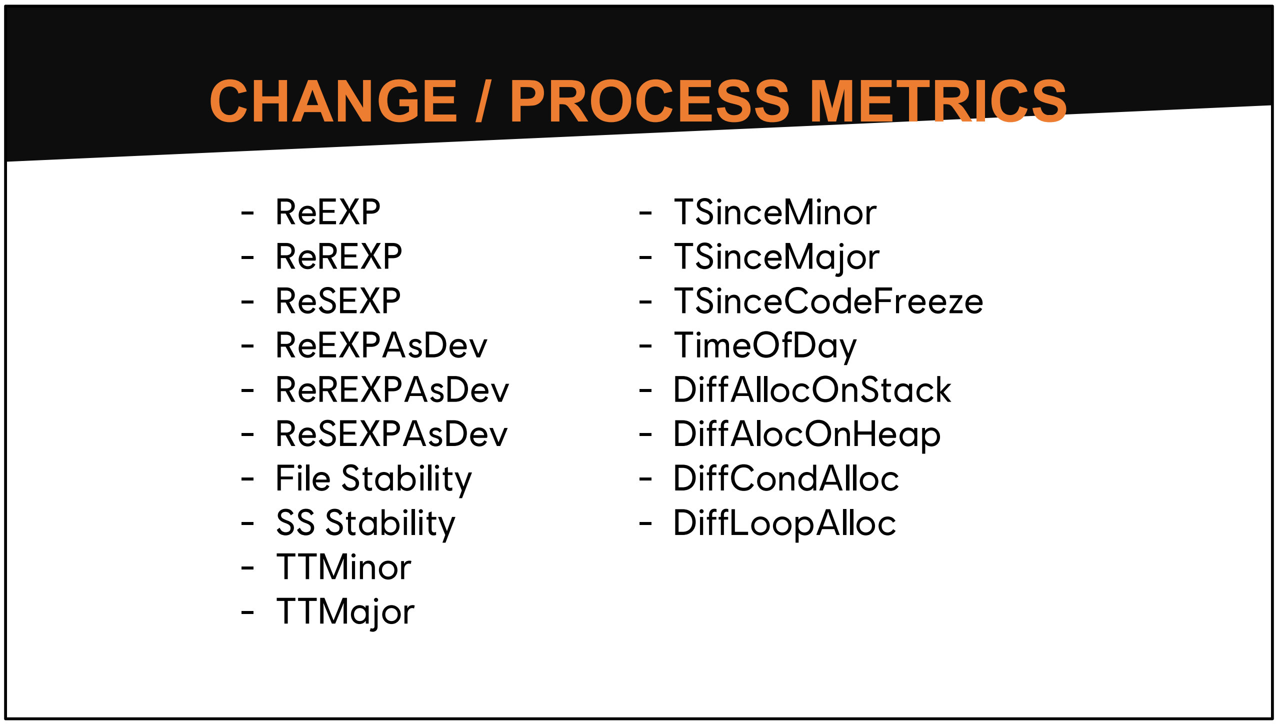
A classical use-case of detecting bug-introducing commit, regardless of the technique, is to build a classifier that can detect them. The field of research when we try to predict them at the commit-level, in opposition to release level, is just-in-time defect introduction prediction. One of the papers that we, and a lot of others, are guided by is this one by Yasutaka et al.



CHANGE / PROCESS METRICS

- NS
- ND
- NF
- Entropy
- LA
- LD
- LT
- NDEV
- AGE
- NUC
- EXP
- REXP
- SEXP

To build the classifier we mine and then use metrics that describe the bug-introducing commits. Classical metrics that are described in this paper include number of systems, number of directories, number of files, entropy of the changes, line added, line deleted, line total, number of devs that touched the files, age of the files, unique changes, experience, recent experience and, subsystem experience.



CHANGE / PROCESS METRICS

- ReEXP
- ReREXP
- ReSEXP
- ReEXPAsDev
- ReREXPAsDev
- ReSEXPAsDev
- File Stability
- SS Stability
- TTMinor
- TTMajor
- TSinceMinor
- TSinceMajor
- TSinceCodeFreeze
- TimeOfDay
- DiffAllocOnStack
- DiffAlocOnHeap
- DiffCondAlloc
- DiffLoopAlloc

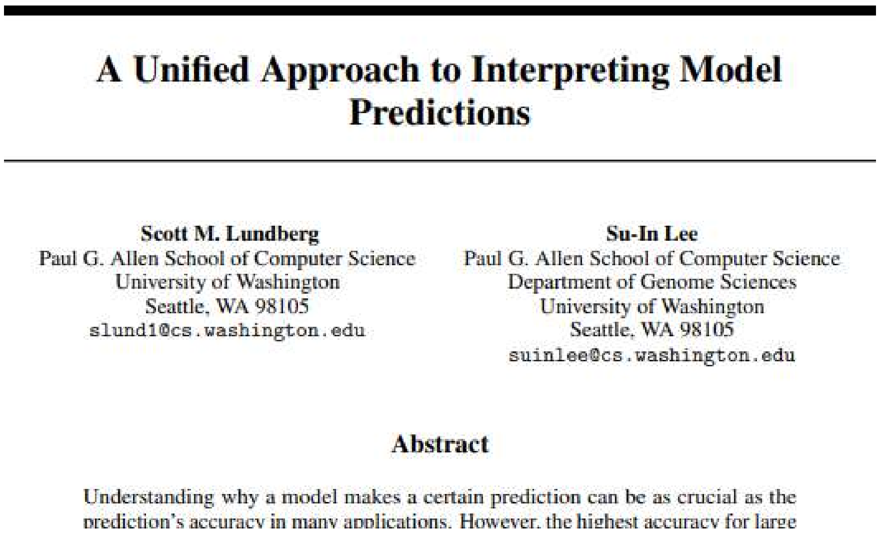
We experimented a lot with this, across dozens of AAA games and dozens of internal tools and found out that a lot of other metrics are yielding significant added accuracy.

For instance, the experiences of the assigned reviewer as reviewer and as developer weight a lot. We also have the file stability, the subsystem stability, the times since and to the last and next minor and majors' releases date. We are also looking at the allocation done on the heap, and the stack. We also do that on loops and behind conditional branches.

We compute all of this metrics based on the fine-grained changes mined on the AST using visitors that compute, or estimate in the case of allocations, the values of the metrics.

Then, we train a classifier on this. You can do linear regression, trees or boosted trees or even go deep-learning on this. Currently, we are using xGBoost.

Explainability


A Unified Approach to Interpreting Model Predictions

Scott M. Lundberg
Paul G. Allen School of Computer Science
University of Washington
Seattle, WA 98105
slund1@cs.washington.edu

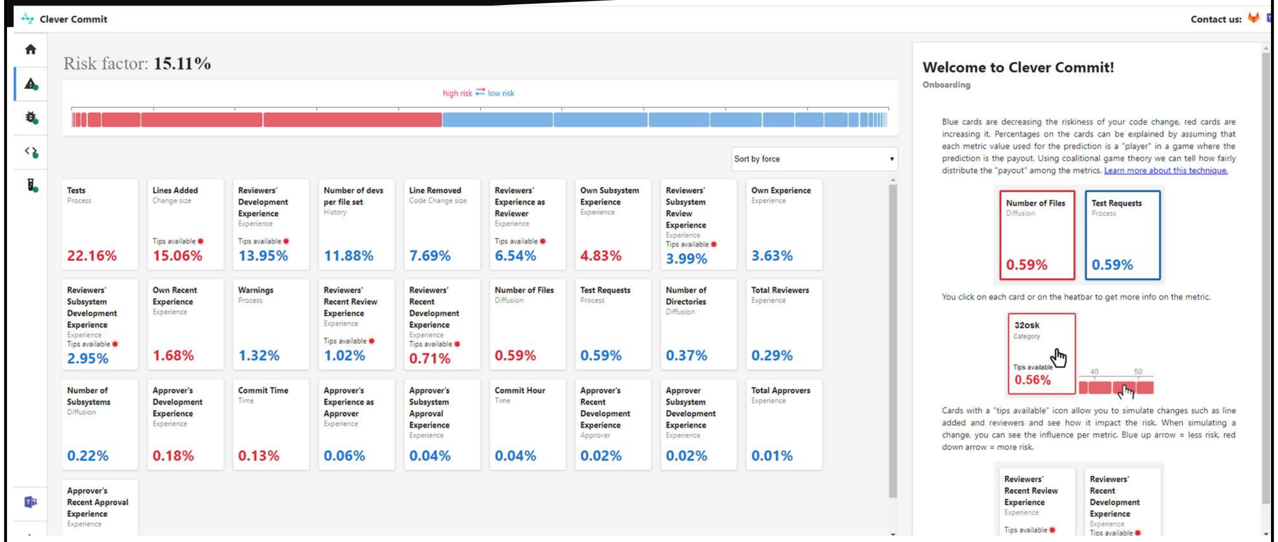
Su-In Lee
Paul G. Allen School of Computer Science
Department of Genome Sciences
University of Washington
Seattle, WA 98105
suinlee@cs.washington.edu

Abstract

Understanding why a model makes a certain prediction can be as crucial as the prediction's accuracy in many applications. However, the highest accuracy for large

This paper, published at NIPS introduced the SHAP values and changes the game for us. SHAP values gives an unified approach to interpreting a model. In other words, for each prediction, you can have the weight of each metric and see how it impacts a given prediction.

Explainability


Clever Commit

Contact us:

Risk factor: **15.11%**

high risk ⇄ low risk

Sort by force

Tests	Process		22.16%
Lines Added	Change size	Tips available ●	15.06%
Reviewers' Development Experience	Experience	Tips available ●	13.95%
Number of devs per file set	History		11.88%
Line Removed	Code Change size		7.69%
Reviewers' Experience as Reviewer	Experience	Tips available ●	6.54%
Own Subsystem Experience	Experience		4.83%
Reviewers' Subsystem Review Experience	Experience	Tips available ●	3.99%
Own Experience	Experience		3.63%
Reviewers' Subsystem Development Experience	Experience	Tips available ●	2.95%
Own Recent Experience	Experience		1.68%
Warnings	Process		1.32%
Reviewers' Recent Review Experience	Experience	Tips available ●	1.02%
Reviewers' Recent Development Experience	Experience	Tips available ●	0.71%
Number of Files	Diffusion		0.59%
Test Requests	Process		0.59%
Number of Directories	Diffusion		0.37%
Total Reviewers	Experience		0.29%
Number of Subsystems	Diffusion		0.22%
Approver's Development Experience	Experience		0.18%
Commit Time	Time		0.13%
Approver's Experience as Approver	Experience		0.06%
Approver's Subsystem Approval Experience	Experience		0.04%
Commit Hour	Time		0.04%
Approver's Recent Development Experience	Approver		0.02%
Approver Subsystem Development Experience	Experience		0.02%
Total Approvers	Experience		0.01%
Approver's Recent Approval Experience	Experience		

Welcome to Clever Commit!

Onboarding

Blue cards are decreasing the riskiness of your code change, red cards are increasing it. Percentages on the cards can be explained by assuming that each metric value used for the prediction is a "player" in a game where the prediction is the payout. Using coalitional game theory we can tell how fairly distribute the "payout" among the metrics. Learn more about this technique.

Number of Files	Diffusion	0.59%
Test Requests	Process	0.59%

You click on each card or on the heatbar to get more info on the metric.

32osk	Category	Tips available	0.56%

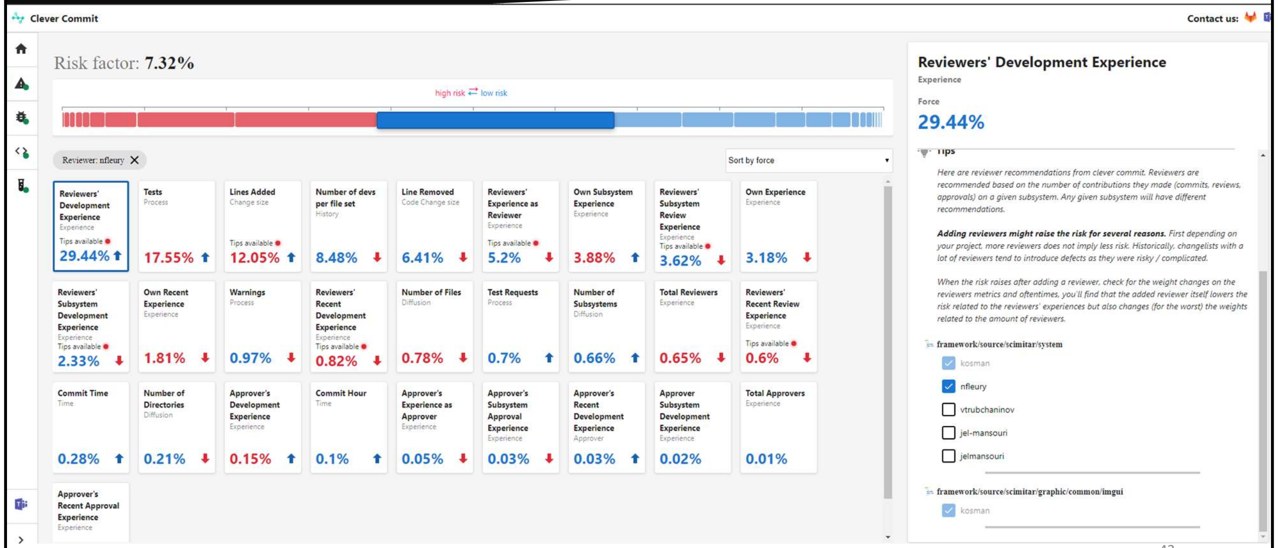
Cards with a "tips available" icon allow you to simulate changes such as line added and reviewers and see how it impact the risk. When simulating a change, you can see the influence per metric. Blue up arrow = less risk, red down arrow = more risk.

Reviewers' Recent Review Experience	Experience	Tips available ●
Reviewers' Recent Development Experience	Experience	Tips available ●

42

We built a tool around the classifier and the SHAP values that developers see when they are preparing their commit; before the commit is sent to the code repository. In this tool, we see all the feature and their weight, either good or bad, towards the riskiness of introducing a new defect shown in the top-left corner.

Explainability



Clever Commit

Contact us:

Risk factor: **7.32%**

high risk ⇄ low risk

Reviewer: nfleury

Sort by force

Reviewers' Development Experience Experience Tips available ● 29.44% ⬆	Tests Process 17.55% ⬆	Lines Added Change size Tips available ● 12.05% ⬆	Number of devs per file set History 8.48% ⬇	Line Removed Code Change size 6.41% ⬇	Reviewers' Experience as Reviewer Experience Tips available ● 5.2% ⬇	Own Subsystem Experience Experience 3.88% ⬆	Reviewers' Subsystem Review Experience Experience Tips available ● 3.62% ⬇	Own Experience Experience 3.18% ⬇
Reviewers' Subsystem Development Experience Experience Tips available ● 2.33% ⬇	Own Recent Experience Experience 1.81% ⬇	Warnings Process 0.97% ⬇	Reviewers' Recent Development Experience Experience Tips available ● 0.82% ⬇	Number of Files Diffusion 0.78% ⬇	Test Requests Process 0.7% ⬆	Number of Subsystems Diffusion 0.66% ⬆	Total Reviewers Experience 0.65% ⬇	Reviewers' Recent Review Experience Experience Tips available ● 0.6% ⬇
Commit Time Time 0.28% ⬆	Number of Directories Diffusion 0.21% ⬇	Approver's Development Experience Experience 0.15% ⬆	Commit Hour Time 0.1% ⬆	Approver's Experience as Approver Experience 0.05% ⬇	Approver's Subsystem Approval Experience Experience 0.03% ⬇	Approver's Recent Development Experience Approver 0.03% ⬆	Approver Subsystem Development Experience Experience 0.02%	Total Approvers Experience 0.01%
Approver's Recent Approval Experience Experience								

Reviewers' Development Experience

Experience

Force

29.44%

Tips

Here are reviewer recommendations from clever commit. Reviewers are recommended based on the number of contributions they made (commits, reviews, approvals) on a given subsystem. Any given subsystem will have different recommendations.

Adding reviewers might raise the risk for several reasons. *First depending on your project, more reviewers does not imply less risk. Historically, changelists with a lot of reviewers tend to introduce defects as they were risky / complicated.*

When the risk raises after adding a reviewer, check for the weight changes on the reviewers metrics and oftentimes, you'll find that the added reviewer itself lowers the risk related to the reviewers' experiences but also changes (for the worst) the weights related to the amount of reviewers.

framework/source/scimitar/system

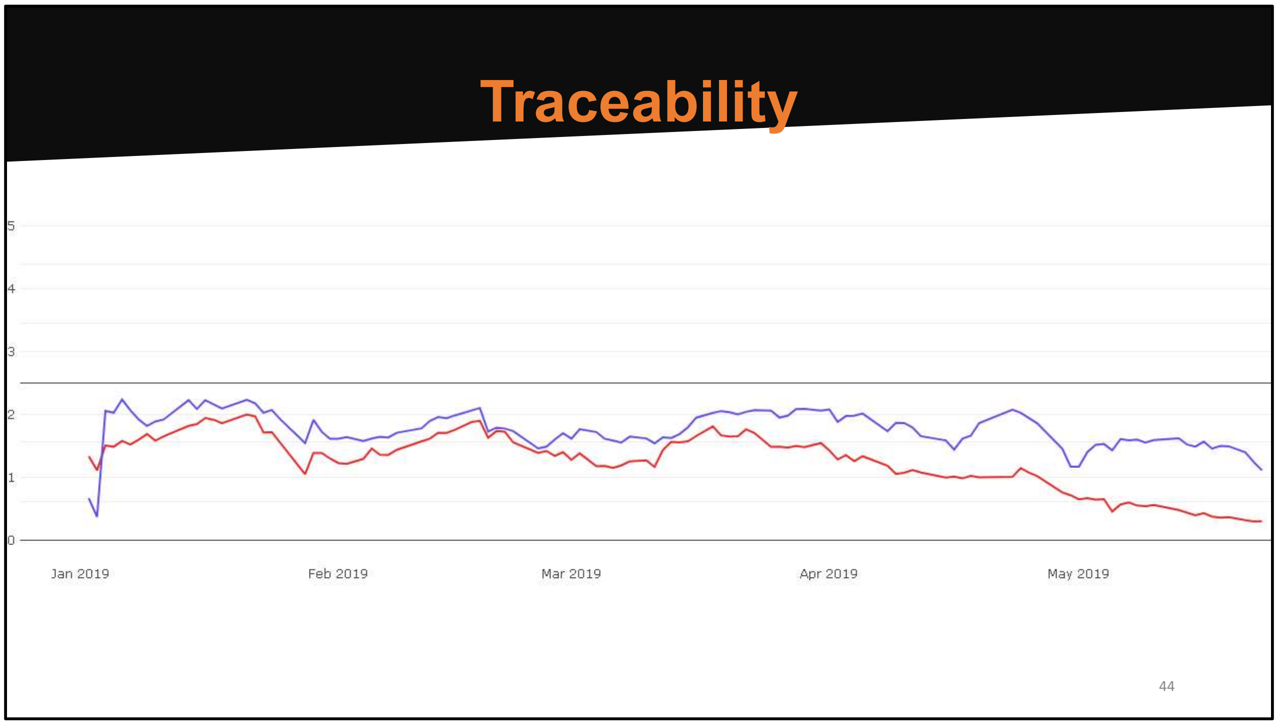
- [✓] kosman
- [✓] nfleury
- [ ] vtrubchaninov
- [ ] jel-mansouri
- [ ] jelmansouri

framework/source/scimitar/graphic/common/imgui

- [✓] kosman

43

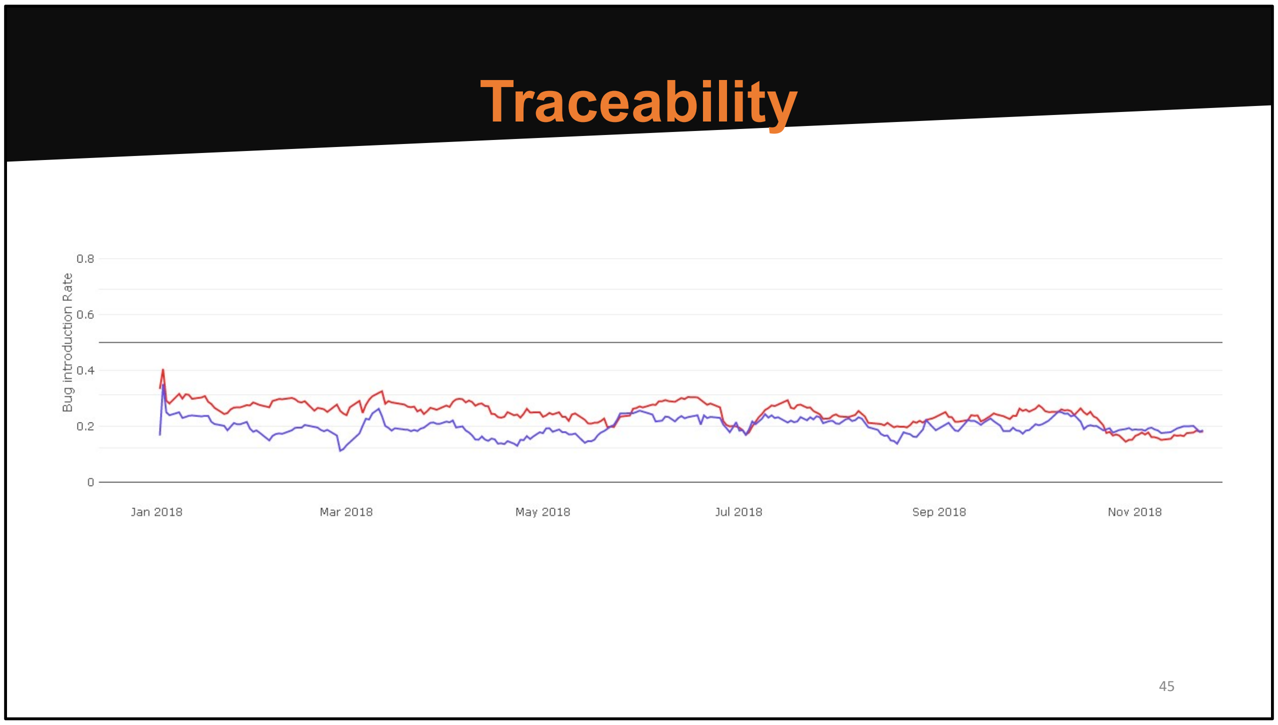
Then, within the tool, developers can tweak their commits by selecting suggested reviewers. The suggestions are based on the amount of contributions, or code ownership, of the region of code modified by the commit at hand. We also offer simulations that modifies the number of lines of code, the complexity and so on. At each simulation, the classifier re-classifies the commit as if the simulations were real and gives a feedback, in form of arrows, to the developers so they know how their simulation impacted the riskiness.



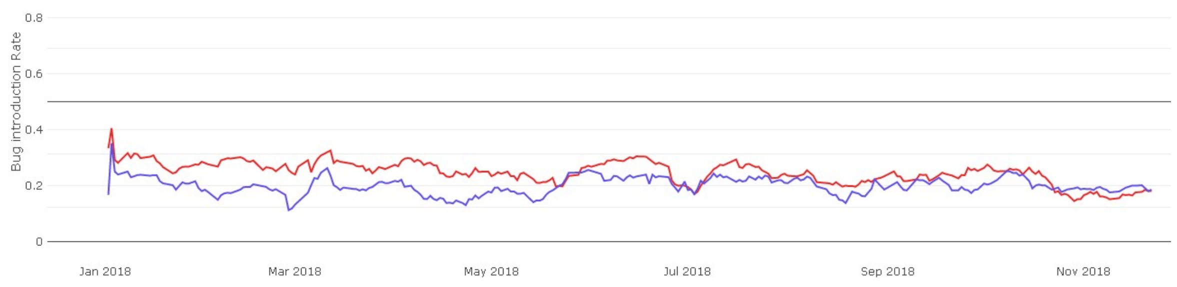
Traceability

Another aspect that we found to be of crucial importance when applying this research in an industrial setting is traceability. In addition to be able to explain a single prediction, you need to be able to see how the classifications are performing over time. We built reporting for this that are accessible to anyone. One of them show the actual bug introduction rate for a given project, in red, and the predicted bug introduction rate in blue.

At the beginning, the lines are following each other very closely but we can see a big difference at the end. This difference is explained by the fact that the bugs are there, they were introduced but they are not found yet.



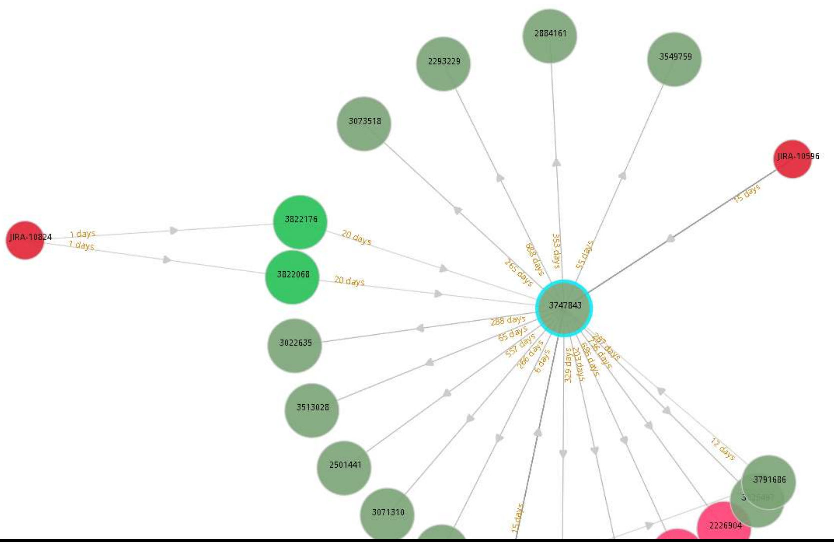
Traceability



45

We can be sure of this by doing classification for long enough. As you can see, the prediction and the reality lines are following each other in the long run.

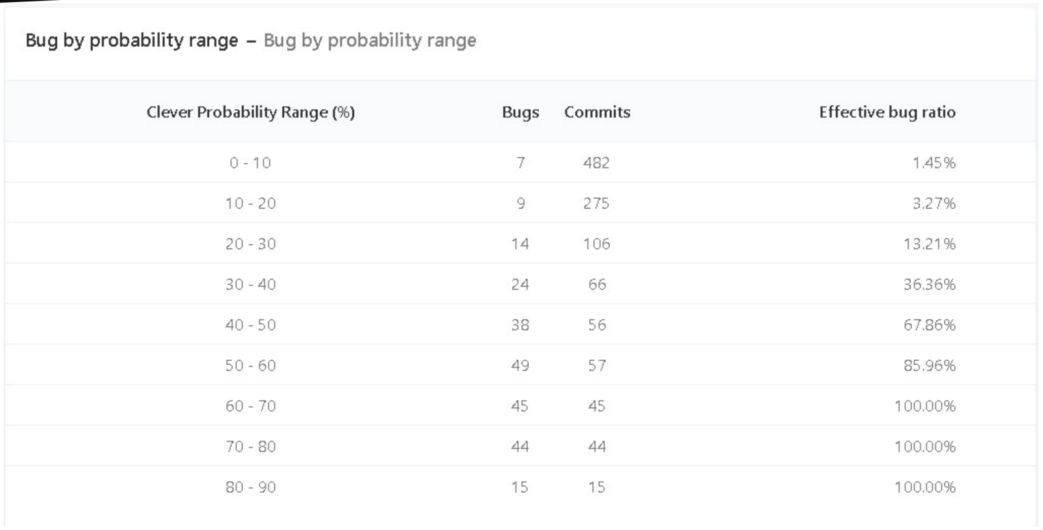
Traceability



46

Still, on traceability, at the commit level, when we classify a commit to be a bug-fix or a bug introducing commit, we built a commit-history graph that developers can explore to be reassured in the classifications. Here, we focus on a given commit and all the commits that modified it afterwards. We also see the jiras that these commits are linked to.

Traceability

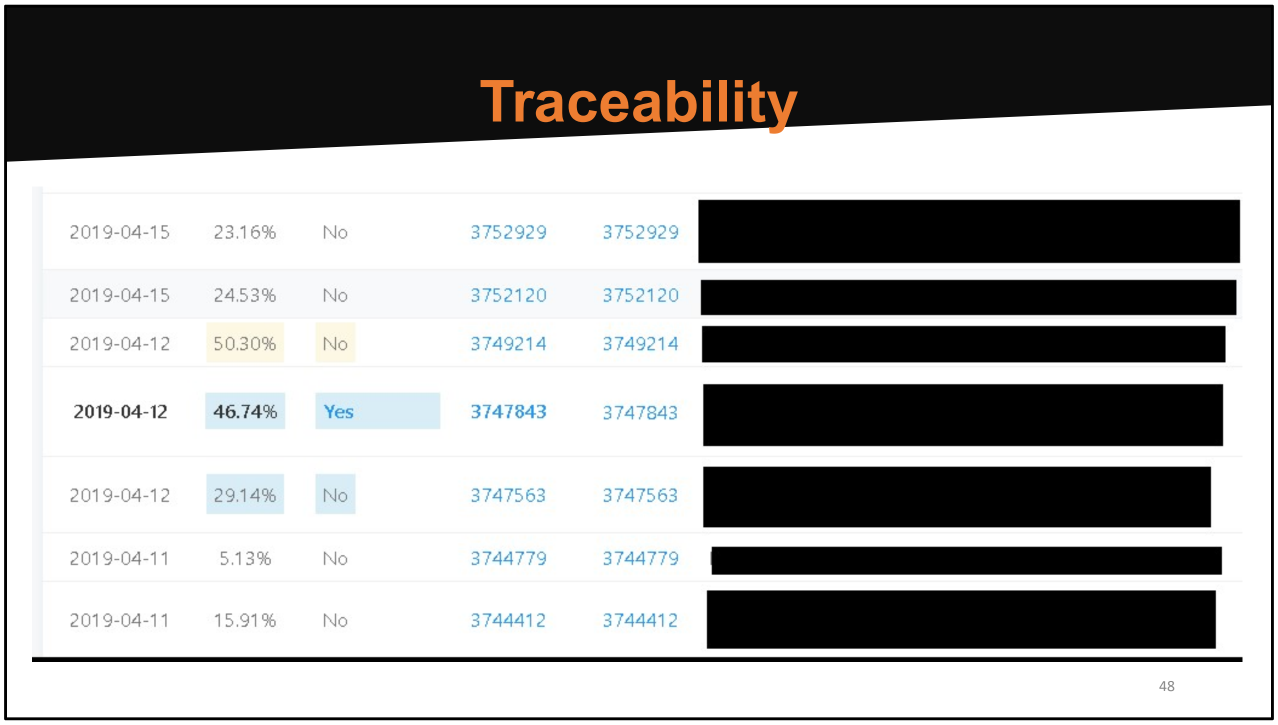


Bug by probability range – Bug by probability range

Clever Probability Range (%)	Bugs	Commits	Effective bug ratio
0 - 10	7	482	1.45%
10 - 20	9	275	3.27%
20 - 30	14	106	13.21%
30 - 40	24	66	36.36%
40 - 50	38	56	67.86%
50 - 60	49	57	85.96%
60 - 70	45	45	100.00%
70 - 80	44	44	100.00%
80 - 90	15	15	100.00%

47

We also found that interpreting the riskiness of a commit to introduce a defect was not trivial for our engineers and managers. In our reporting, we can see, over-time the actual performances of each range of prediction. Here, we have a 36% effective bug ratio or true positives when the riskiness is between 30% and 40%. Above 60% the true-positive rate is 100%.

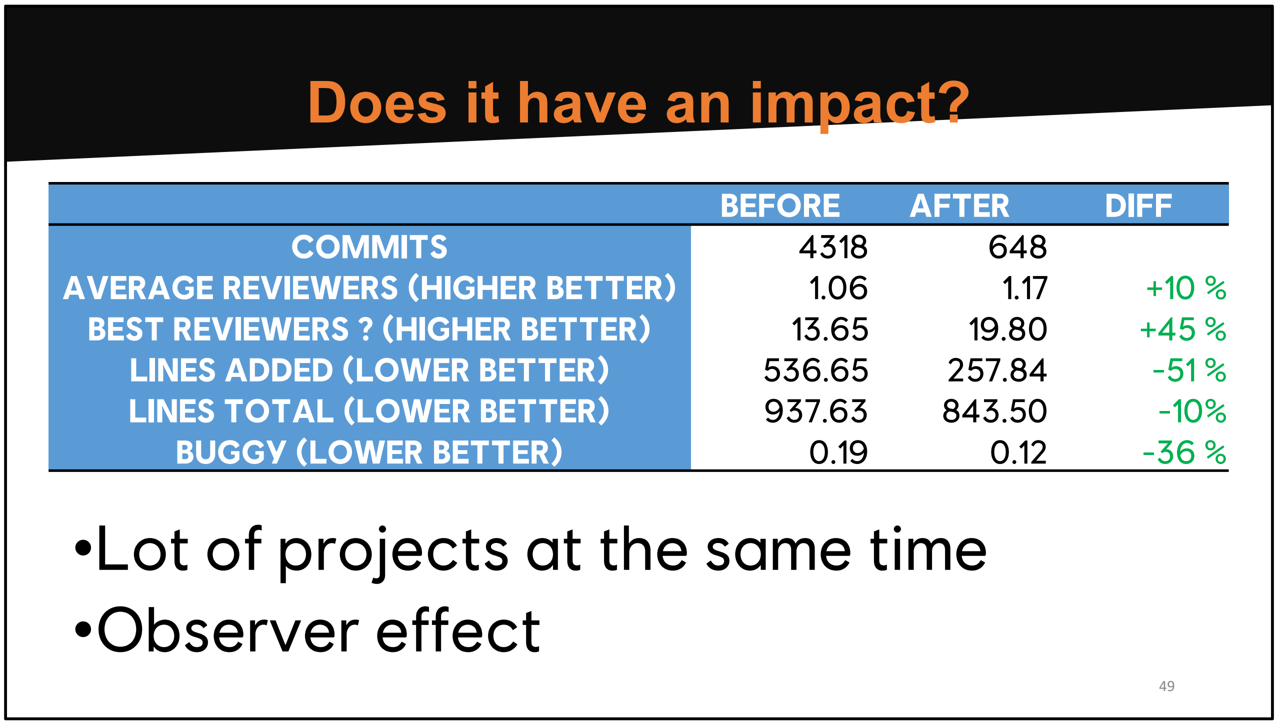


Traceability

2019-04-15	23.16%	No	3752929	3752929	[redacted]
2019-04-15	24.53%	No	3752120	3752120	[redacted]
2019-04-12	50.30%	No	3749214	3749214	[redacted]
2019-04-12	**46.74%**	**Yes**	**3747843**	3747843	[redacted]
2019-04-12	29.14%	No	3747563	3747563	[redacted]
2019-04-11	5.13%	No	3744779	3744779	[redacted]
2019-04-11	15.91%	No	3744412	3744412	[redacted]

48

Because defects are not identified instantly, we make sure to display it as soon as they are identified. Here, for the prediction at 46.74% we found a bug and the *YES* will redirect you to the jira ticket.



Does it have an impact?

	BEFORE	AFTER	DIFF
COMMITS	4318	648	
AVERAGE REVIEWERS (HIGHER BETTER)	1.06	1.17	+10 %
BEST REVIEWERS ? (HIGHER BETTER)	13.65	19.80	+45 %
LINES ADDED (LOWER BETTER)	536.65	257.84	-51 %
LINES TOTAL (LOWER BETTER)	937.63	843.50	-10%
BUGGY (LOWER BETTER)	0.19	0.12	-36 %

- Lot of projects at the same time
- Observer effect

49

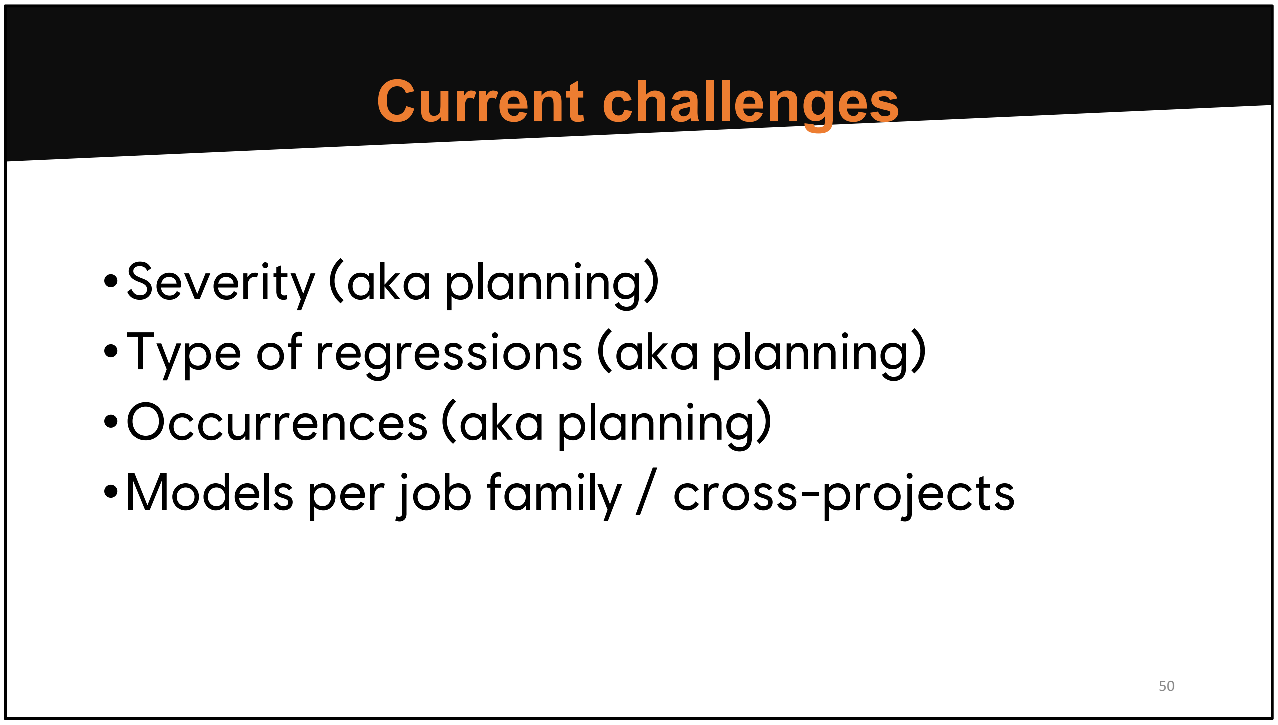
The big question then is: does it have an impact? We did a comparison study before and after the deployment of the tool on one project where the team didn't grow and other factors such as the time to release did not changed significantly as the game was planned for many years later.

We found a 36% reduction in bug introduction rate.

We started from 19%, which is very aligned to what you can find in big open-source systems and dropped down to 12%. It also came with some additional benefits such as the experience of the selected reviewers and reduction in the number of lines.

While we could control from some external threats while doing this experimentation. All the threats were not assessed. For one, a lot of initiatives are aiming to enhance the developers experience and productivity. Enhancement to the build-systems, new tests being written and so on.

We also have to wonder about the observer effect. Did the tool actually lead to the reduction or the developers, knowing that a tool would be reading their code, were more careful?



Current challenges

- Severity (aka planning)
- Type of regressions (aka planning)
- Occurrences (aka planning)
- Models per job family / cross-projects

50

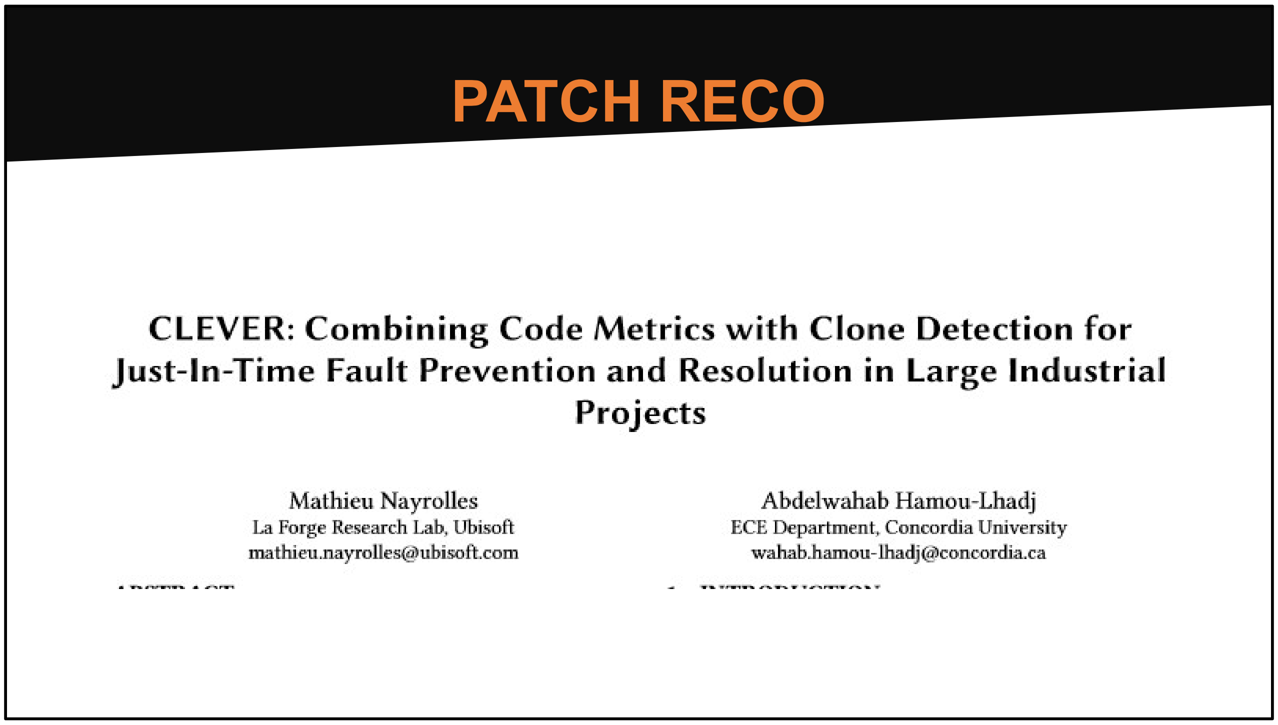
The current challenges we are trying to assess now are to build more classifier to further help the developers to smartly invest their time refining a code-contribution. We are looking at predicting the severity of the potential defects, the type of regressions: is it a crash, a gameplay bug, an online misshape? We are also looking in predicting the number of occurrences the crash could have if introduced.

All of this are to help planning our effort. In the end, a lot of efforts must be put in ironing the last bugs and some of them are more important than other. More important because a lot of players experience them or because they cause a final crash. These new classifiers will, hopefully, help us with that.

We also found that sharing models between similar projects works ok. But while we are using cross-projects model we are currently experiencing for job-family models that are shared across projects. The variance in metrics for the subsystems handling sound are very different from the subsystems handling animations or physics. If this is validated, in the coming months, we'll be operating the same job-related models across different projects.

If you are working on solving these challenges, don’ t hesitate to reach out and we can maybe work on this together.

PATCH SUGGESTIONS



PATCH RECO

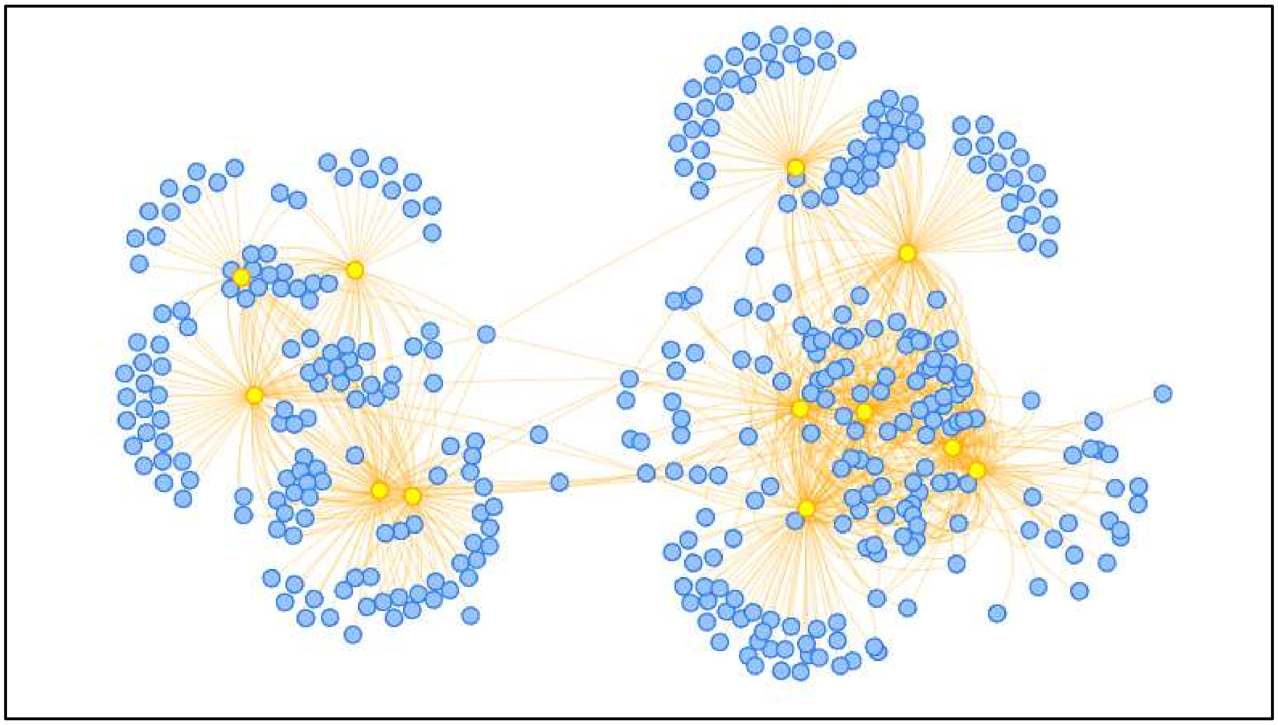
CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects

Mathieu Nayrolles
La Forge Research Lab, Ubisoft
mathieu.nayrolles@ubisoft.com

Abdelwahab Hamou-Lhadj
ECE Department, Concordia University
wahab.hamou-lhadj@concordia.ca

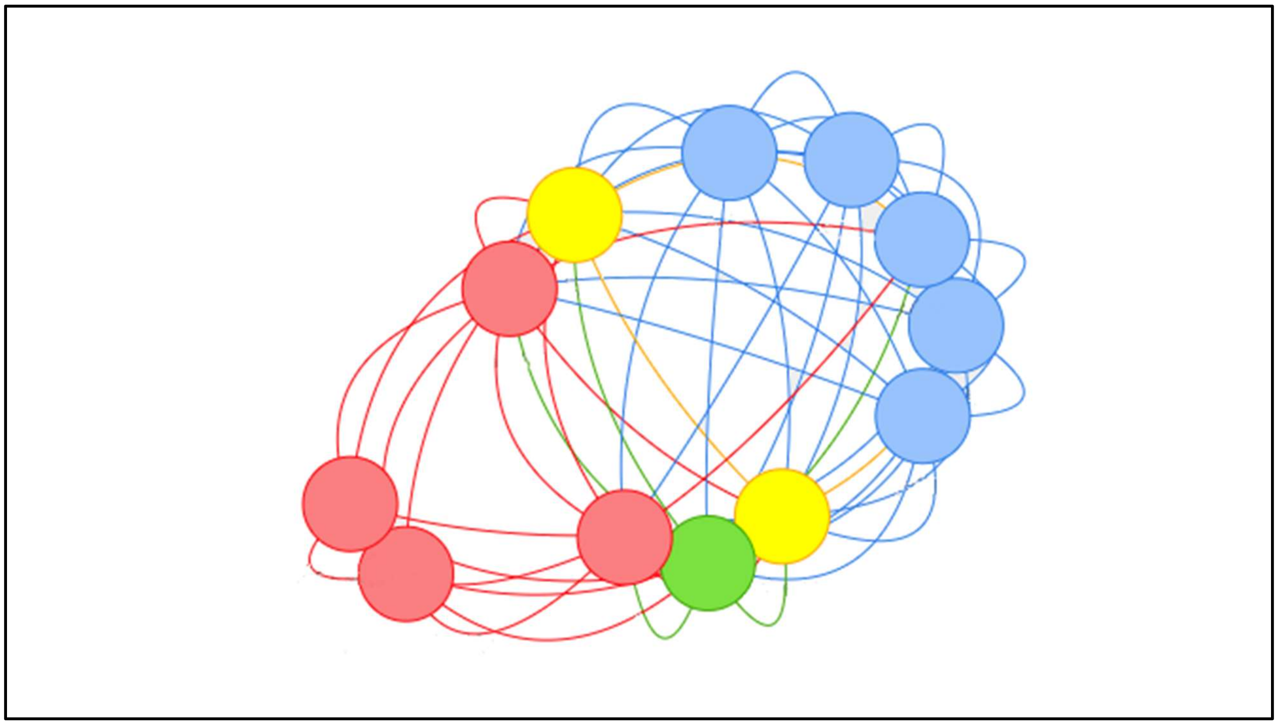
The next step to help developers, after predicting the riskiness of introducing a defect and explaining why is to propose a code-change. These patch suggestions or automated program repairs if done efficiently could greatly enhance developer productivity.

We have proposed our own attempt at this at MSR'18.



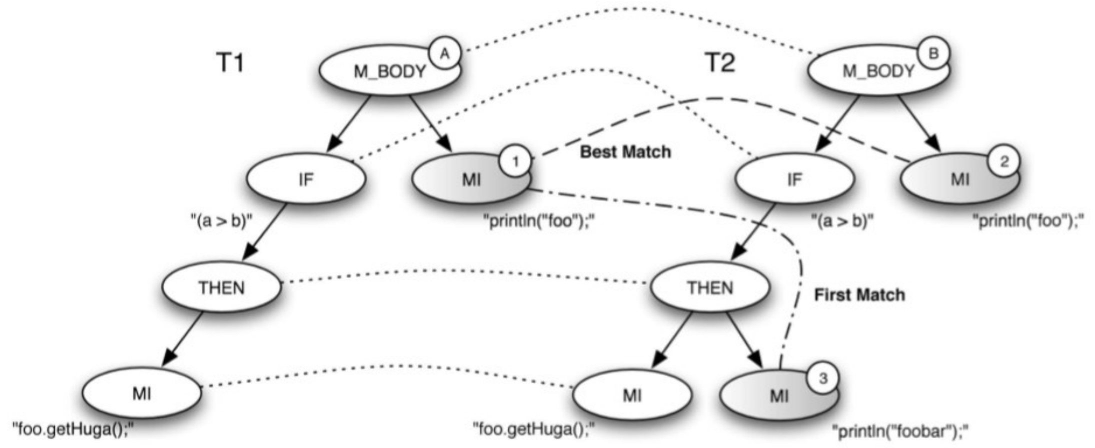
It was based on clustering project that looks like each other in terms of code but also in terms of dependency. The relational behind this is that if two projects are using the same dependencies, then they are likely to be opened to the same issues.

Here, in yellow are the projects while the dependencies are in blue.

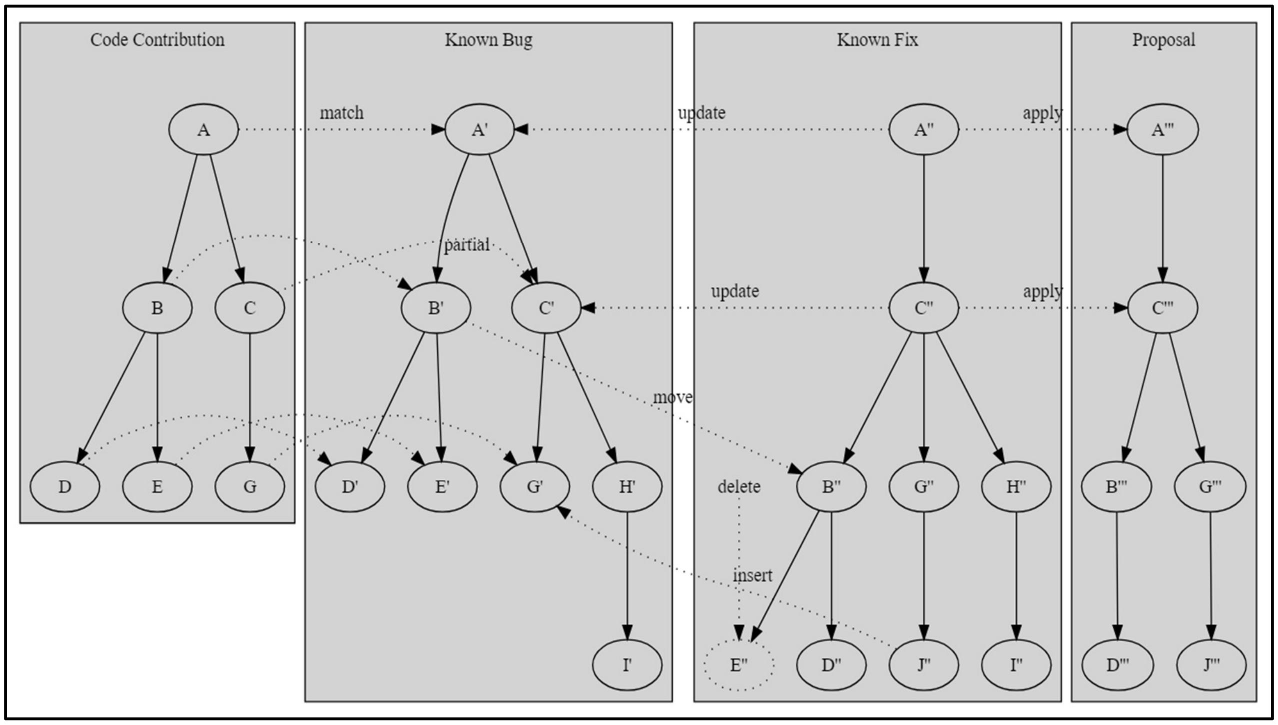


Within the clusters, for each code change we did a clone comparison against abstracted known bugs and proposed the fixes applied to the developer.

CHANGE DETECTION



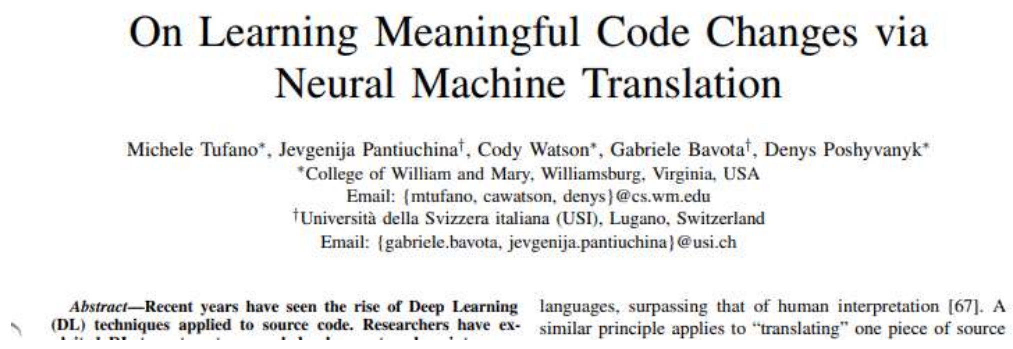
We further refined it by having the fixes merged to the code of the developer using the algorithms described by Fluri and al in the TSE paper I talked about earlier.



To give you an idea of how that works, if a given code contribution A-B-C-D-E-G matches a known bug A'-B'-C'-D'-E'-G'-H'-I' that was fixed by deleting E', updating C' and A' and inserting J'. We take these changes and apply them to the code proposition to get A''-C''-B''-G''-D''-J''.

We are still using this approach, but it is not live for developer of the company to see. We have a lot of scalability issues when comparing to all the known bug of a cluster and doing so for all the contributions as they are submitted. Despite validations by experts we found that the approach was yielding uncomfortable number of false-positives when we tried to apply it company-wide. False positives, for automated program repair are extremely worrisome as they destroy the confidence of the developers. It is likely that if the first few code-change recommendations are of poor quality, the developers won't come back to it.

PATCH RECO


On Learning Meaningful Code Changes via Neural Machine Translation

Michele Tufano*, Jevgenija Pantiuchina†, Cody Watson*, Gabriele Bavota†, Denys Poshyvanyk*
*College of William and Mary, Williamsburg, Virginia, USA
Email: {mtufano, cawatson, denys}@cs.wm.edu
†Università della Svizzera italiana (USI), Lugano, Switzerland
Email: {gabriele.bavota, jevgenija.pantiuchina}@usi.ch

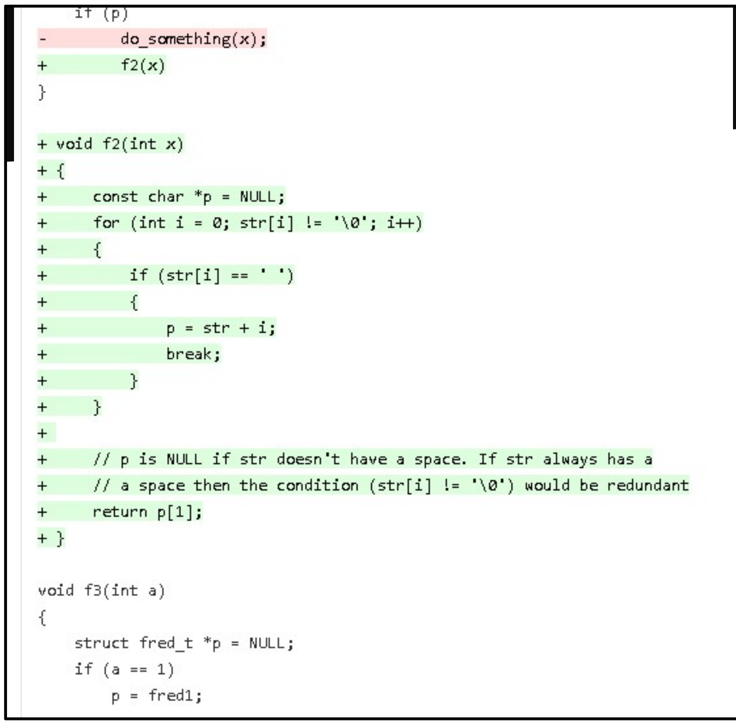
Abstract—**Recent years have seen the rise of Deep Learning (DL) techniques applied to source code. Researchers have ex-**

languages, surpassing that of human interpretation [67]. A similar principle applies to "translating" one piece of source

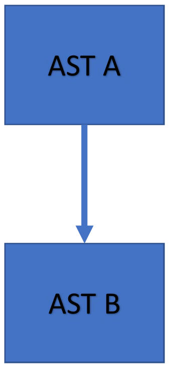
SapFix, Angelix, Hercules, Prophet, Darjeeling, …

In the meantime, a lot of approaches have been published on this topic. One of the current trends is to attack the problem using deep learning and more precisely machine learning translation. A lot of significant examples from academia and industry, using deep-learning or not, have been proposed such as SapFix, Angelix, Hercules and so on.

While these approaches are all very interesting and indeed scalable and accurate, we found that have several flaws that prevent their adoptions; at least in video-games.

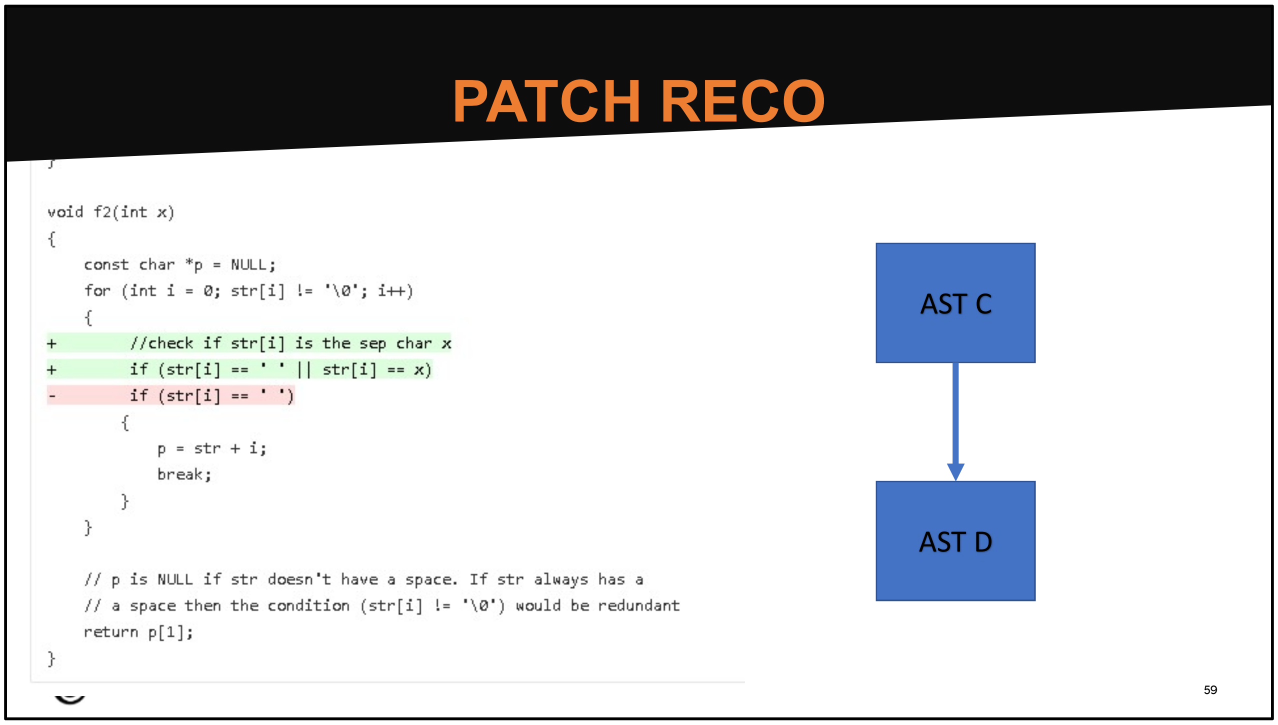


```
    if (p)
-       do_something(x);
+       f2(x)
}

+ void f2(int x)
+ {
+     const char *p = NULL;
+     for (int i = 0; str[i] != '\0'; i++)
+     {
+         if (str[i] == ' ')
+         {
+             p = str + i;
+             break;
+         }
+     }
+
+     // p is NULL if str doesn't have a space. If str always has a
+     // a space then the condition (str[i] != '\0') would be redundant
+     return p[1];
+ }

void f3(int a)
{
    struct fred_t *p = NULL;
    if (a == 1)
        p = fred1;
```



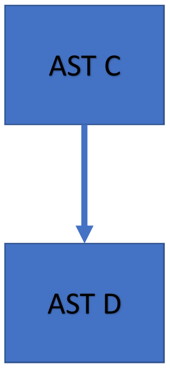
58

To illustrate my point, let's look at this simple example where we have a method extraction. This diff, at the AST level would be represented by an AST transformation from A to B.



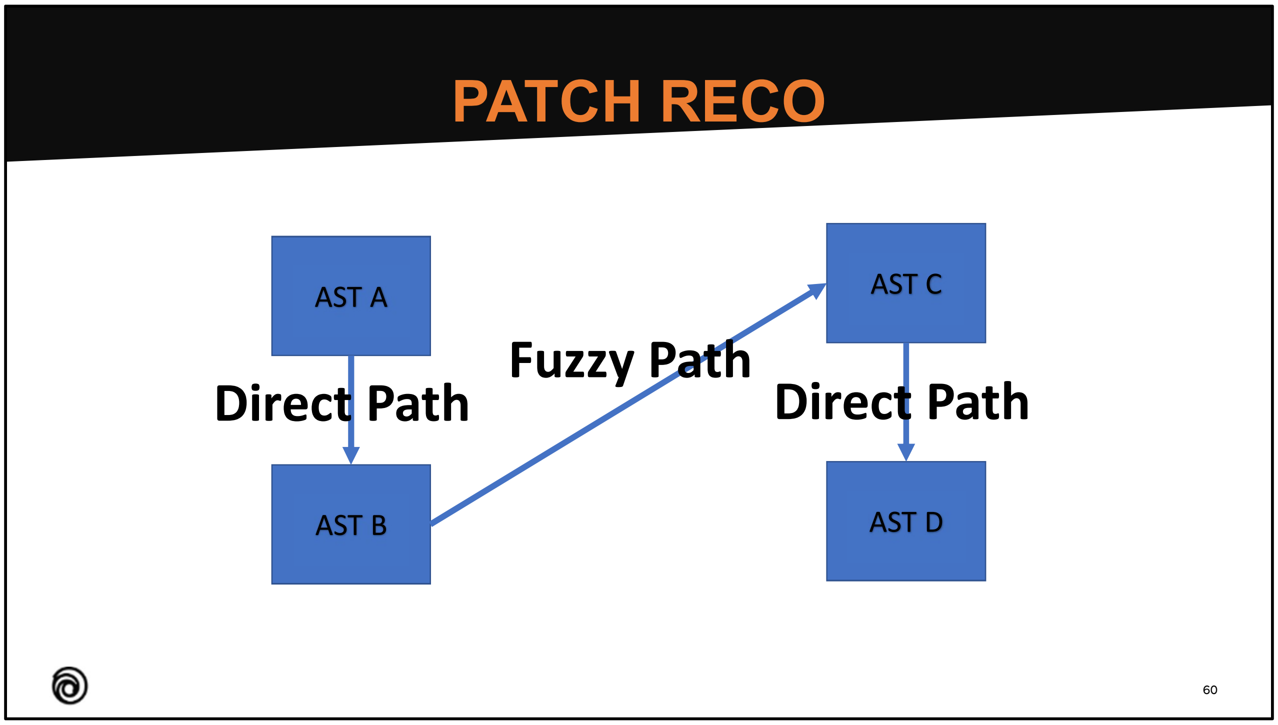
PATCH RECO

```
void f2(int x)
{
    const char *p = NULL;
    for (int i = 0; str[i] != '\0'; i++)
    {
+       //check if str[i] is the sep char x
+       if (str[i] == ' ' || str[i] == x)
-       if (str[i] == ' ')
        {
            p = str + i;
            break;
        }
    }

    // p is NULL if str doesn't have a space. If str always has a
    // a space then the condition (str[i] != '\0') would be redundant
    return p[1];
}
```



59

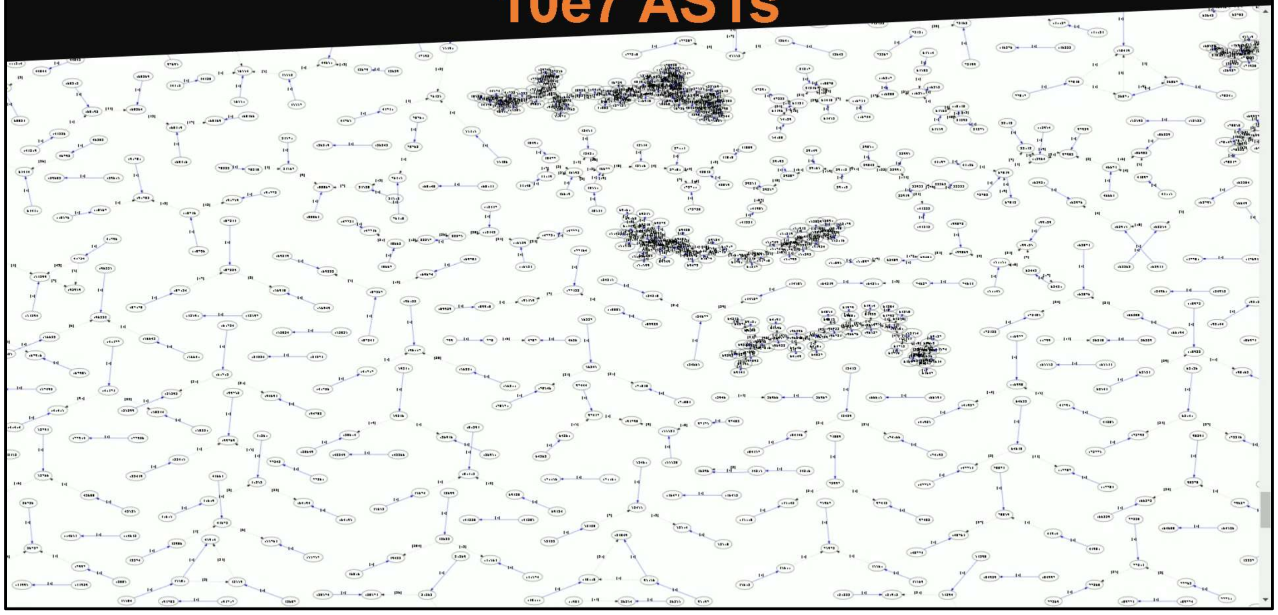
Then, in f2, a bug-fix is made. At the AST level, it would be a transformation from C to D. Each ast is the ast of the code change only and not the whole file.



PATCH RECO

We now have, still at the AST level, what we named direct paths and fuzzy paths. A NMT trained on classical code change would most likely miss the fuzzy path and when shown AST A. In addition, current approaches only consider that a fix is one commit when, in our dataset, we found that a lot of fixes are chains of fixes that are applied to buggy-commits. These chains are incrementally building up to a complete fix that is performant and safe.

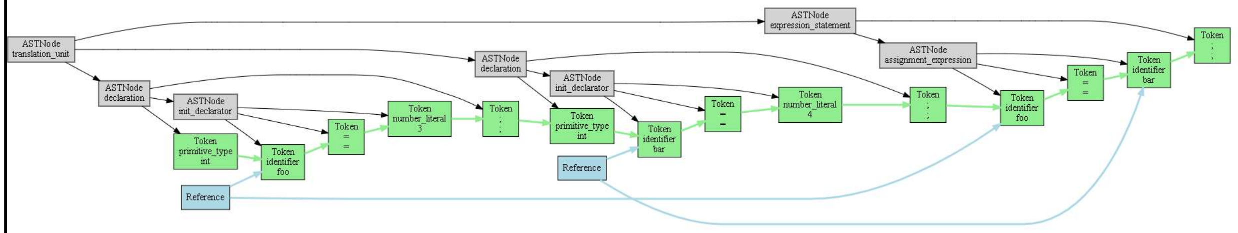
10e7 ASTs



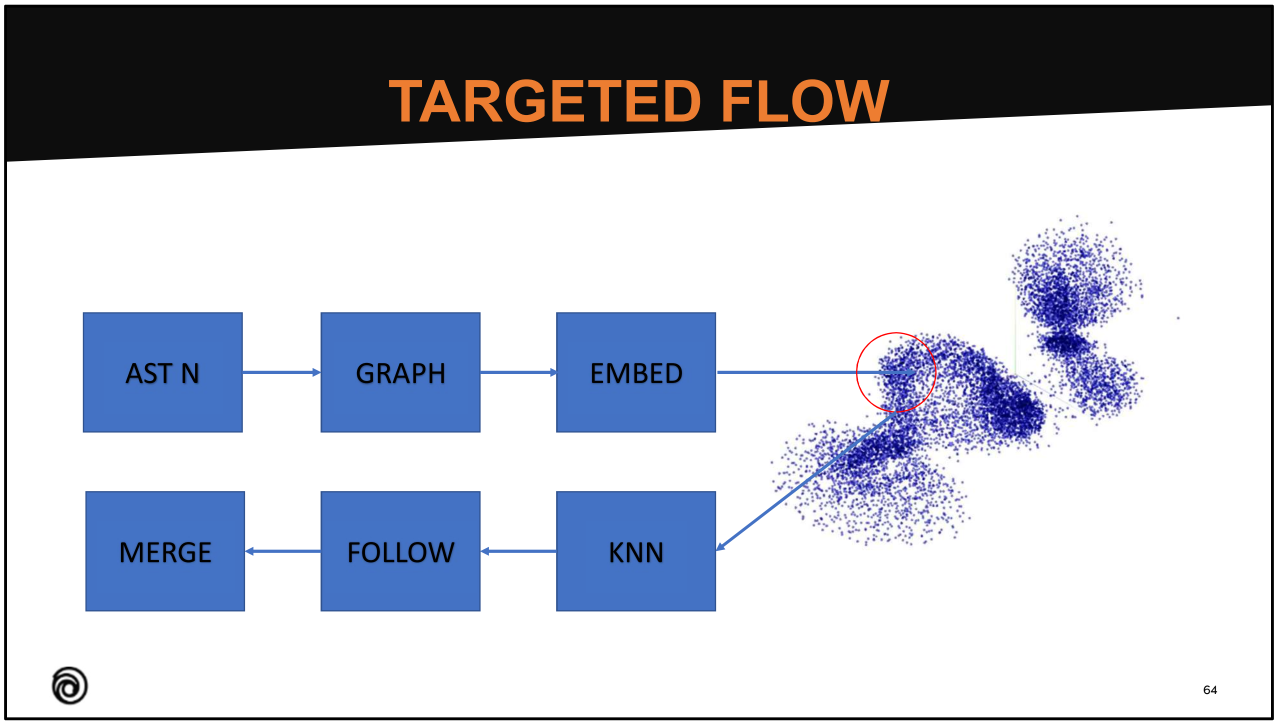
Here you can see a small part in our 10e7 ASTs dataset were some fixes are simple A to B and other involves a lot more work. Both direct and fuzzy paths are represented here.

// Video in the blog-post at laforge.ubisoft.com

ENCODING

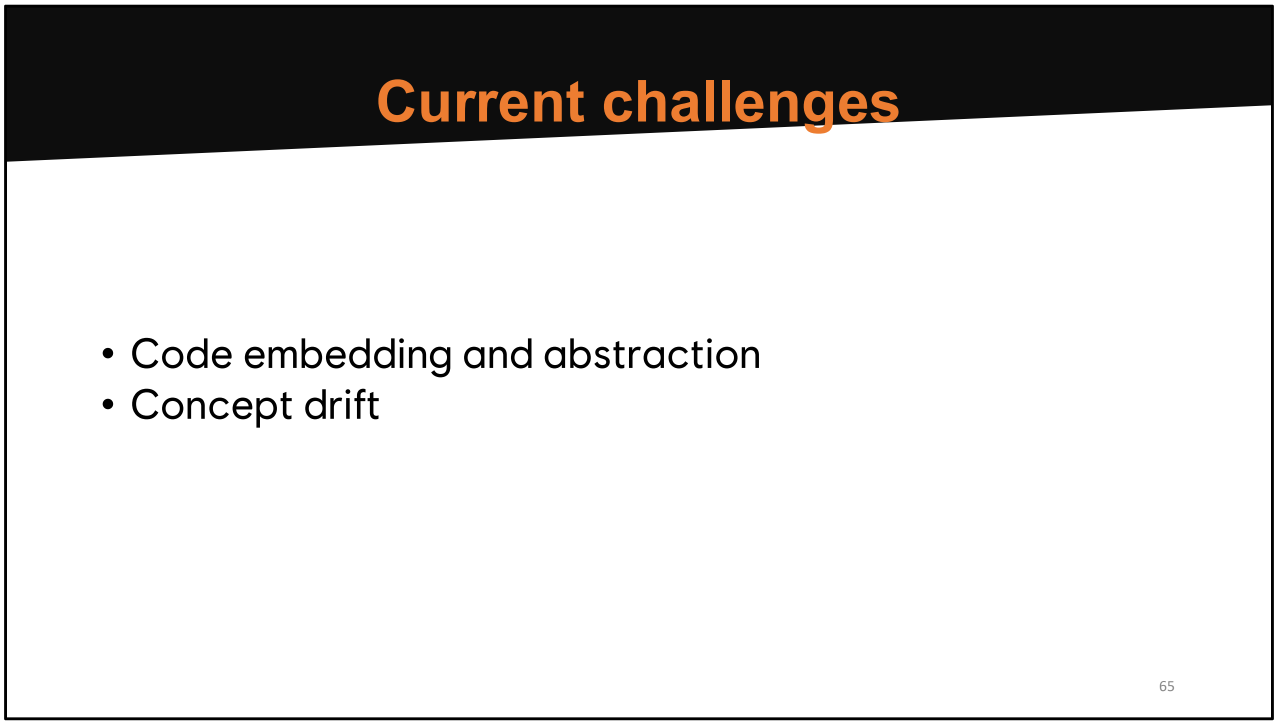


What we are trying to do now, is to encode our ASTs using deep-learning encoder. The AST are vanillas but we do add links between declaration and usage to create a graph.



TARGETED FLOW

Then, the approach would work like so where we have an AST, transform it into a graph and embed it. Then, in the latent space our our deep-learning encoder, we can do a KNN and find known bugs that are similar to the proposed AST. It operates somehow like a near-miss deep-learning clone finder that only focuses on bug-introducing commit. Then, we can follow the fix-chains we found and apply them to the proposed AST. Merging the fix-chain recursively is still done using Fluri and al work. We hope to be able to report soon on this in the framework of a proper scientific publication.



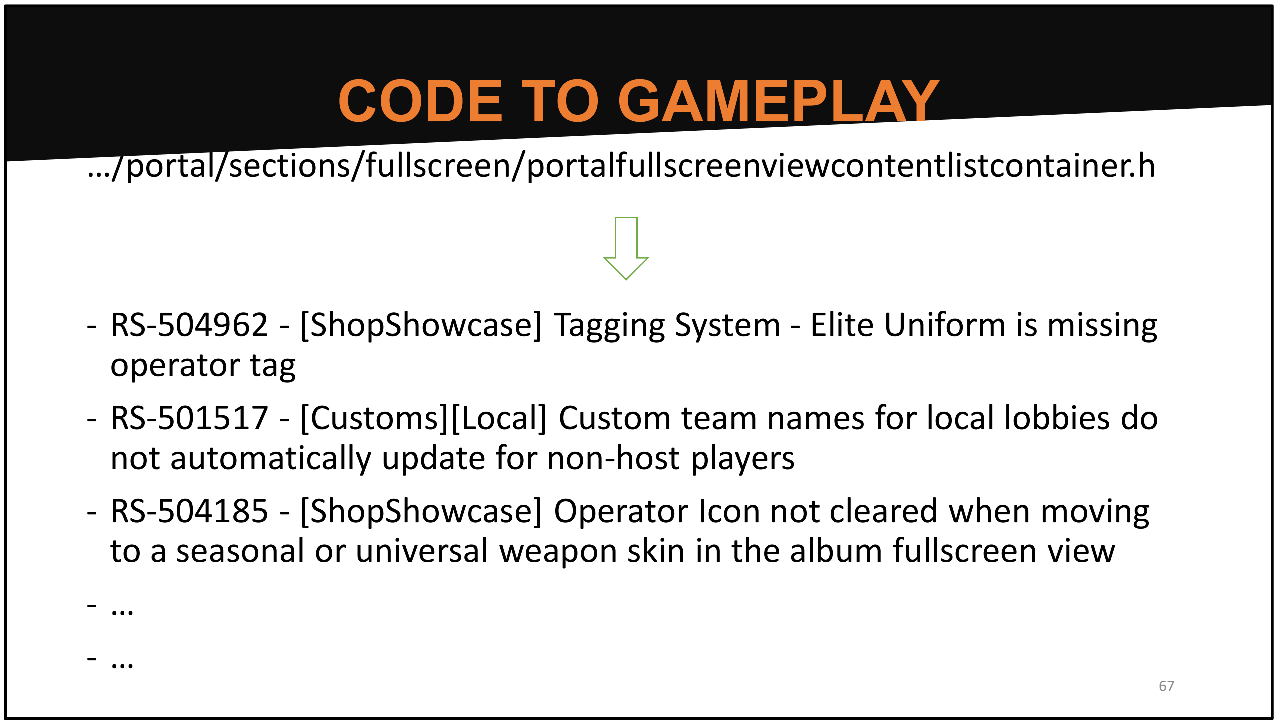
Current challenges

- Code embedding and abstraction
- Concept drift

65

The current challenges we face, and would be glad to collaborate on, w.r.t automated program repair are code-embedding and abstraction. In more details we are experimenting on technics to abstract and then embed code efficiently w.r.t to our use-case in computing the similarity between a code-change and past bugs. Another challenge we face, and that, to the best of our knowledge is faced by other approaches proposed by academia and industry alike is concept drift. We found that fixes that were valid some years ago are not valid anymore. Either because the language was upgraded or the platforms we target changed.

MANUAL TESTS



CODE TO GAMEPLAY

…/portal/sections/fullscreen/portalfullscreenviewcontentlistcontainer.h

⇩

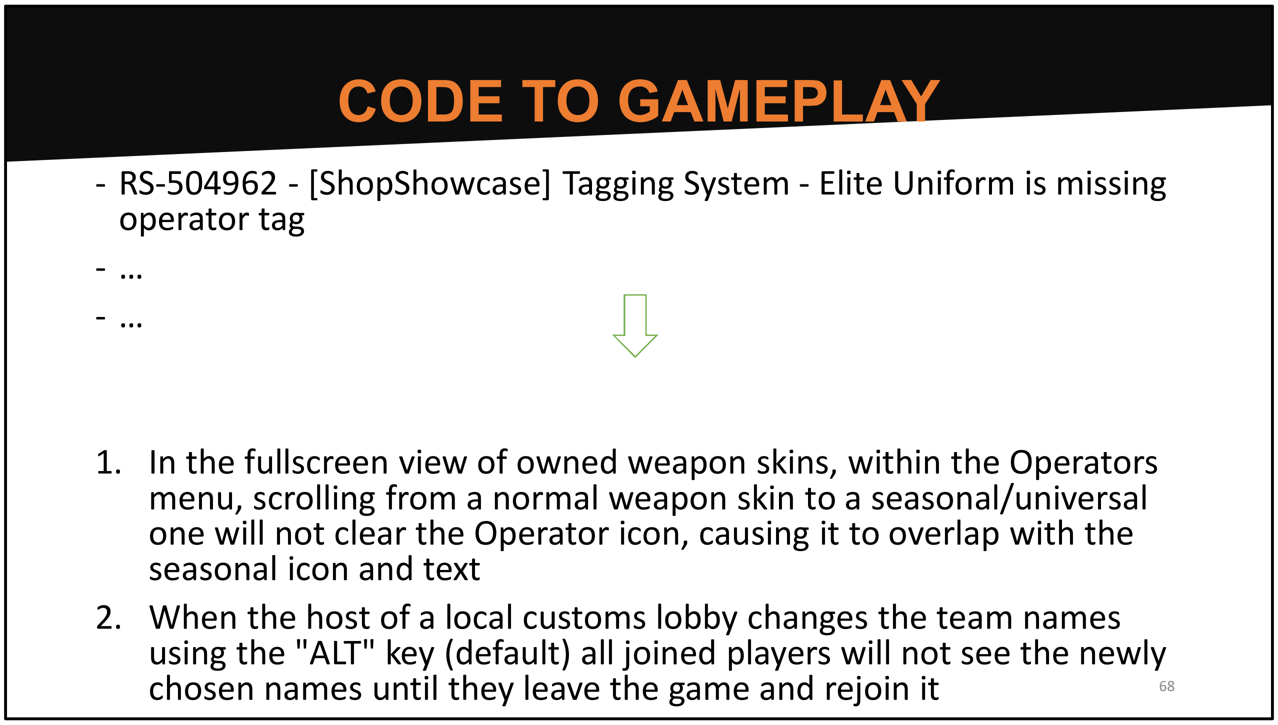
- RS-504962 - [ShopShowcase] Tagging System - Elite Uniform is missing operator tag
- RS-501517 - [Customs][Local] Custom team names for local lobbies do not automatically update for non-host players
- RS-504185 - [ShopShowcase] Operator Icon not cleared when moving to a seasonal or universal weapon skin in the album fullscreen view
- …
- …

One other area we are investing is manual tests. It can sound a bit strange to be speaking about manual tests at ASE but … we are trying to automatically support our testers. Producing video games requires a lot of manual testing. Our manual tests do more than QA, they are also assessing the game mechanics, the playability, the engagement and a lot of other metrics that we deeply care about.

That being said, when they test a build for potential regressions that automated tests could have missed, it’s not easy to know on which piece of the 60-hours game they should focus on. And 60 hours is only the solo part, then you have infinite possibility for the multiplayer mode.

In this part of our work, we are summarizing past regressions that were introduced when the code changed by a commit was changed prior to that commit. It works by selecting the sentences in the jiras linked to buggy commit-regions.

For instance here, when a modification is done to a given region of …/portal/sections/fullscreen/portalfullscreenviewcontentlistcontainer.h, we find a lot of linked jiras. While this could be useful in itself, we go a step further with the summarization.



CODE TO GAMEPLAY

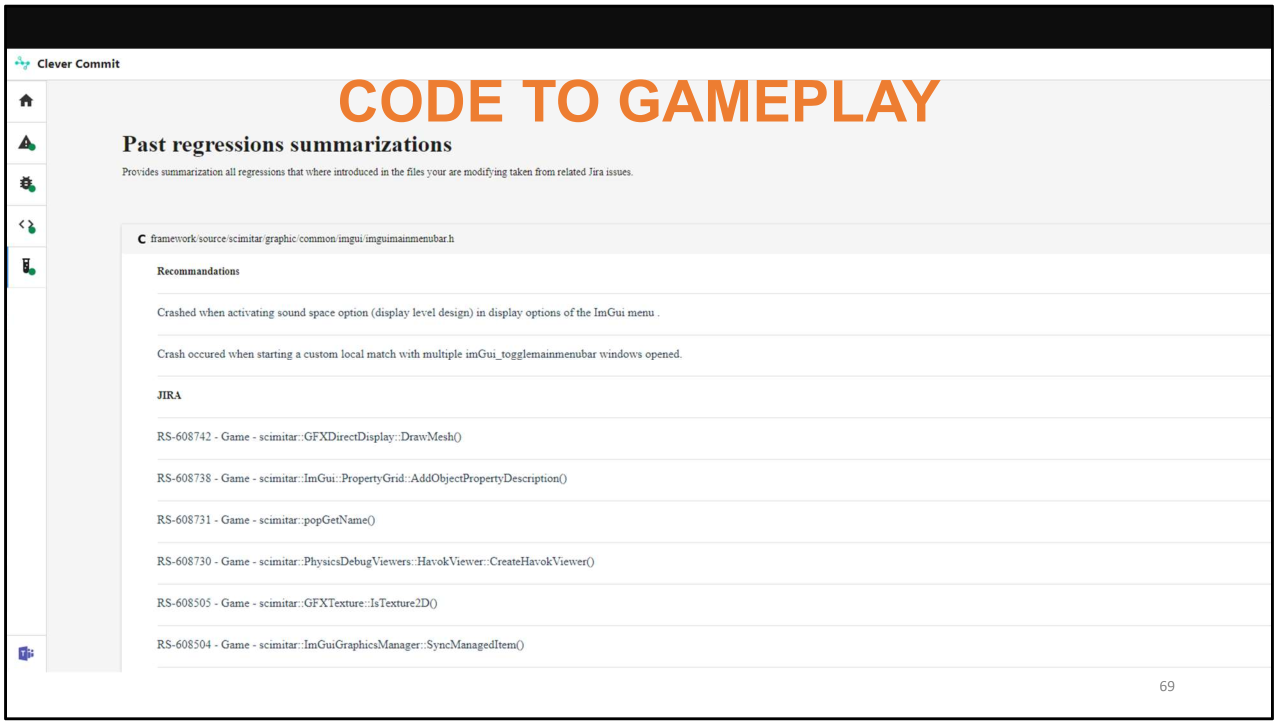
- RS-504962 - [ShopShowcase] Tagging System - Elite Uniform is missing operator tag
- …
- …

⇩

1. In the fullscreen view of owned weapon skins, within the Operators menu, scrolling from a normal weapon skin to a seasonal/universal one will not clear the Operator icon, causing it to overlap with the seasonal icon and text
2. When the host of a local customs lobby changes the team names using the "ALT" key (default) all joined players will not see the newly chosen names until they leave the game and rejoin it

68

Then, the jiras are summarized in a few sentences. In practice it works very well and inform QA testers on where to focus their efforts by translating code-changes into high level gameplay functionality that could be broken. It also helps developer to investigate potential unforeseen coupling between their modification and functionalities they didn't intended to modify.



CODE TO GAMEPLAY

Past regressions summarizations

Provides summarization all regressions that where introduced in the files your are modifying taken from related Jira issues.

C framework/source/scimitar/graphic/common/imgui/imguimainmenubar.h

Recommandations

Crashed when activating sound space option (display level design) in display options of the ImGui menu .

Crash occured when starting a custom local match with multiple imGui_togglemainmenubar windows opened.

JIRA

RS-608742 - Game - scimitar::GFXDirectDisplay::DrawMesh()

RS-608738 - Game - scimitar::ImGui::PropertyGrid::AddObjectPropertyDescription()

RS-608731 - Game - scimitar::popGetName()

RS-608730 - Game - scimitar::PhysicsDebugViewers::HavokViewer::CreateHavokViewer()

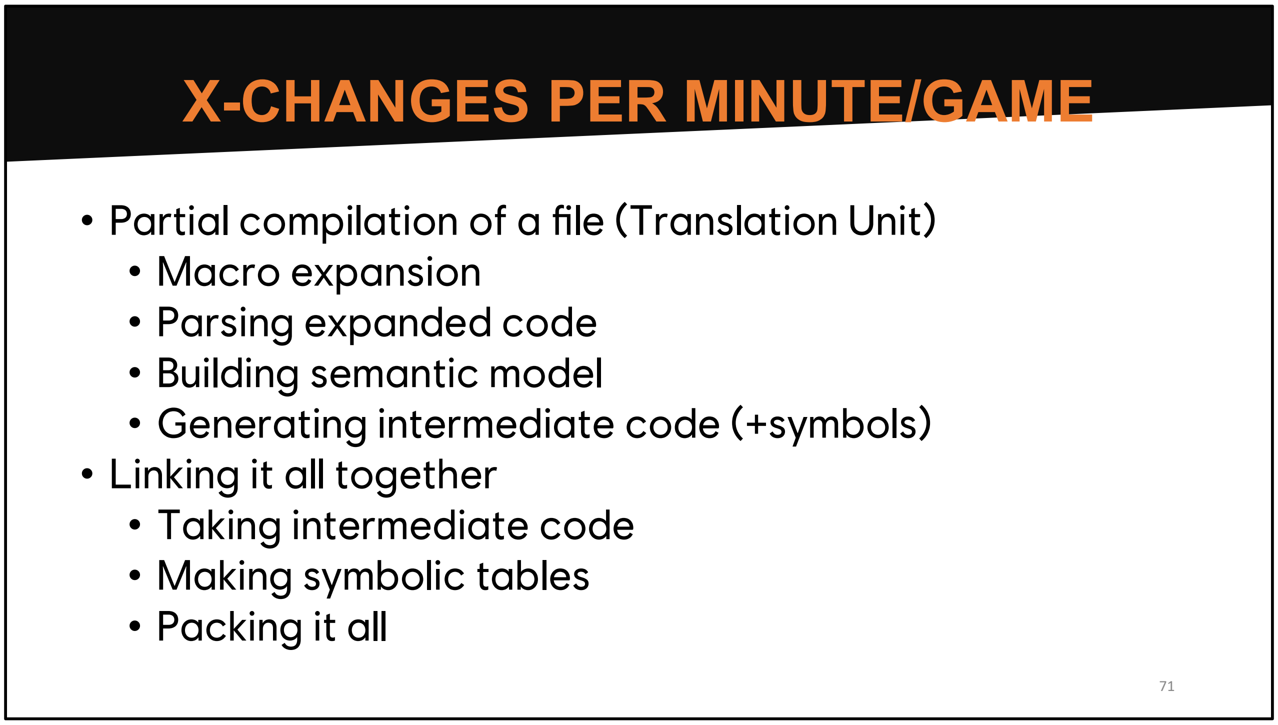
RS-608505 - Game - scimitar::GFXTexture::IsTexture2D()

RS-608504 - Game - scimitar::ImGuiGraphicsManager::SyncManagedItem()

69

It’s part of the same tool as the one that shows and explains the riskiness I presented before.

CI

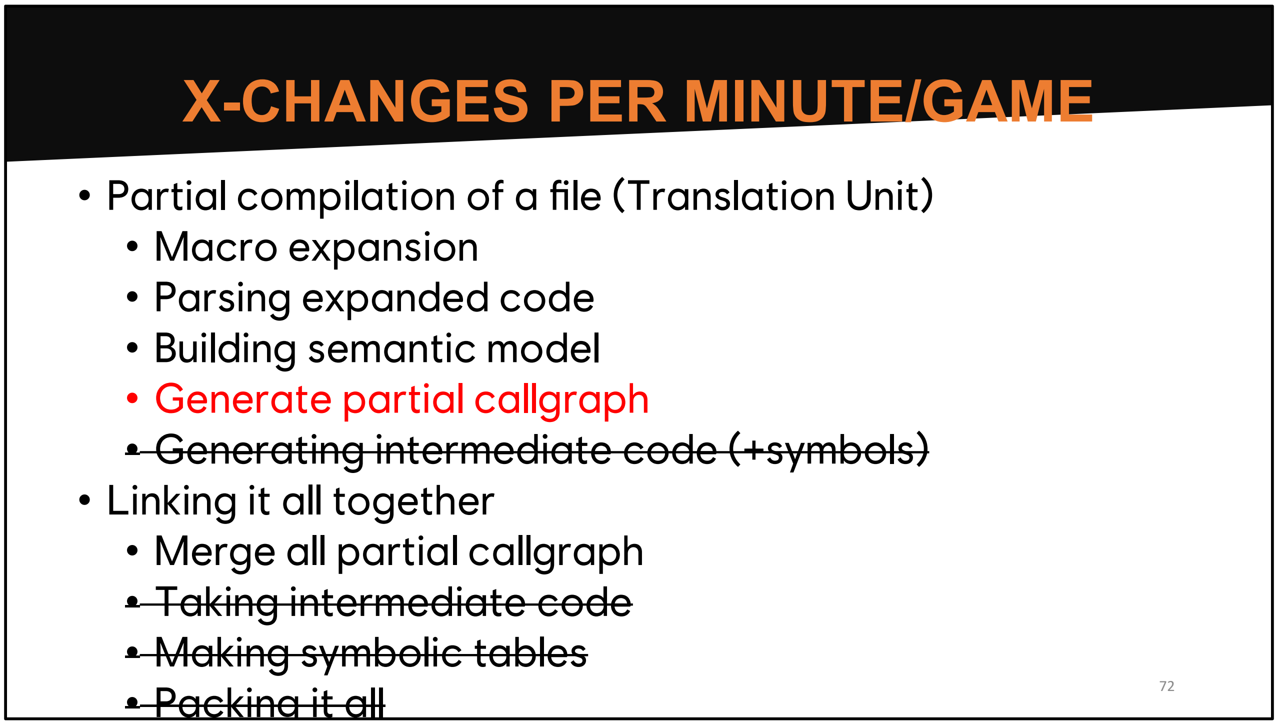


X-CHANGES PER MINUTE/GAME

- Partial compilation of a file (Translation Unit)
  - Macro expansion
  - Parsing expanded code
  - Building semantic model
  - Generating intermediate code (+symbols)
- Linking it all together
  - Taking intermediate code
  - Making symbolic tables
  - Packing it all

71

Because we have several changes per minute per game when we are reaching the productions stages, we are also investigating how to optimize our CI process. We are particularly interested in orchestrating automated tests so the first tests ran are the one the most likely to fail. To so, we are inserting ourselves in the classical steps of the compilation of a cpp codebase.



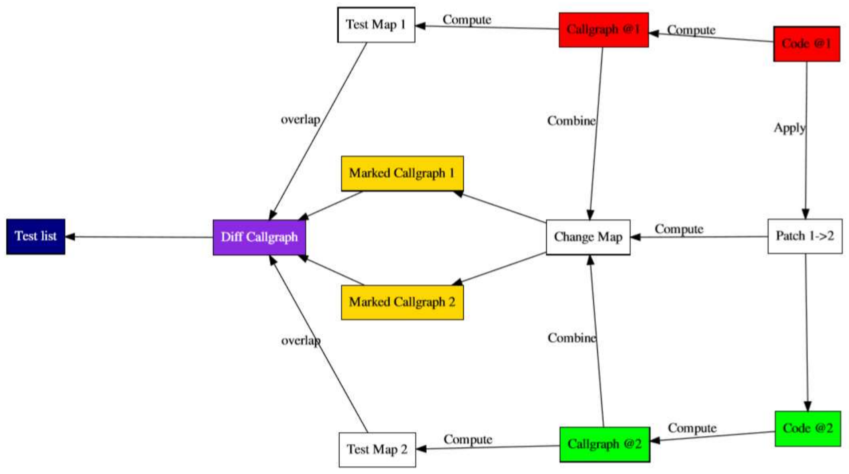
X-CHANGES PER MINUTE/GAME

- Partial compilation of a file (Translation Unit)
  - Macro expansion
  - Parsing expanded code
  - Building semantic model
  - Generate partial callgraph
  - ~~Generating intermediate code (+symbols)~~
- Linking it all together
  - Merge all partial callgraph
  - ~~Taking intermediate code~~
  - ~~Making symbolic tables~~
  - ~~Packing it all~~

72

Once the semantic model is built, we try to generate a partial callgraph that represents what has changed w.r.t to the code to build.

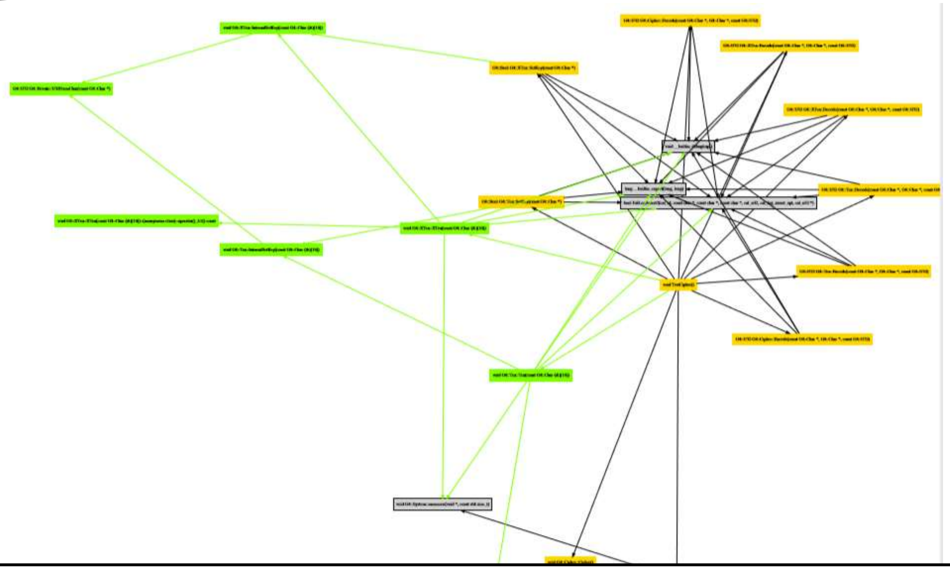
TEST ORCHESTRATION



73

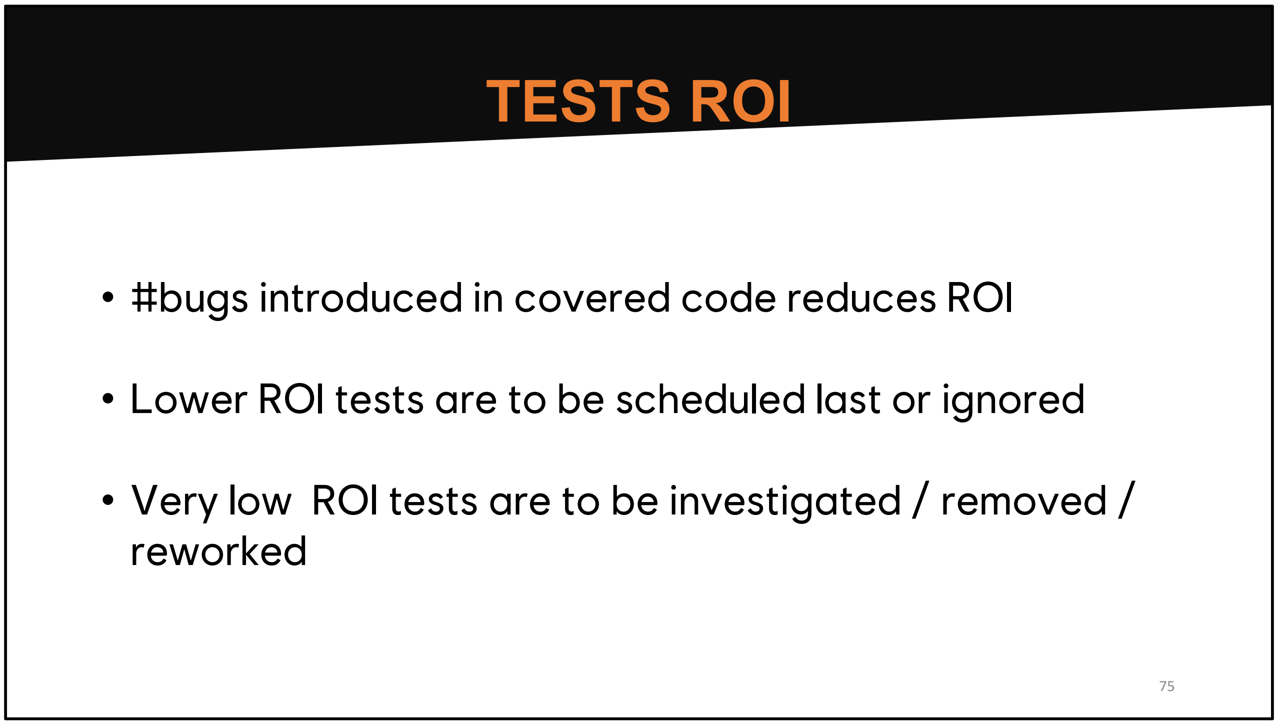
Because we are analyzing also the test code, we are able to tag tests can reach the changed code by using classical static analysis. What we are really after here is the static coverage of automated tests so we can the one that cover the changes first.

TEST ORCHESTRATION



74

In the end, we can see at the method call level, what changes and which tests we should run first. We intended to use machine learning technics to further reduce the amount of test to be ran but our initial experimentations found that the suggested test-list is small enough that it would not make sense to invest in re-orchestrating that list using machine learning. Note that we still advice to run the full-test suite regardless of the static coverage as the static coverage of automated test is far from perfect.

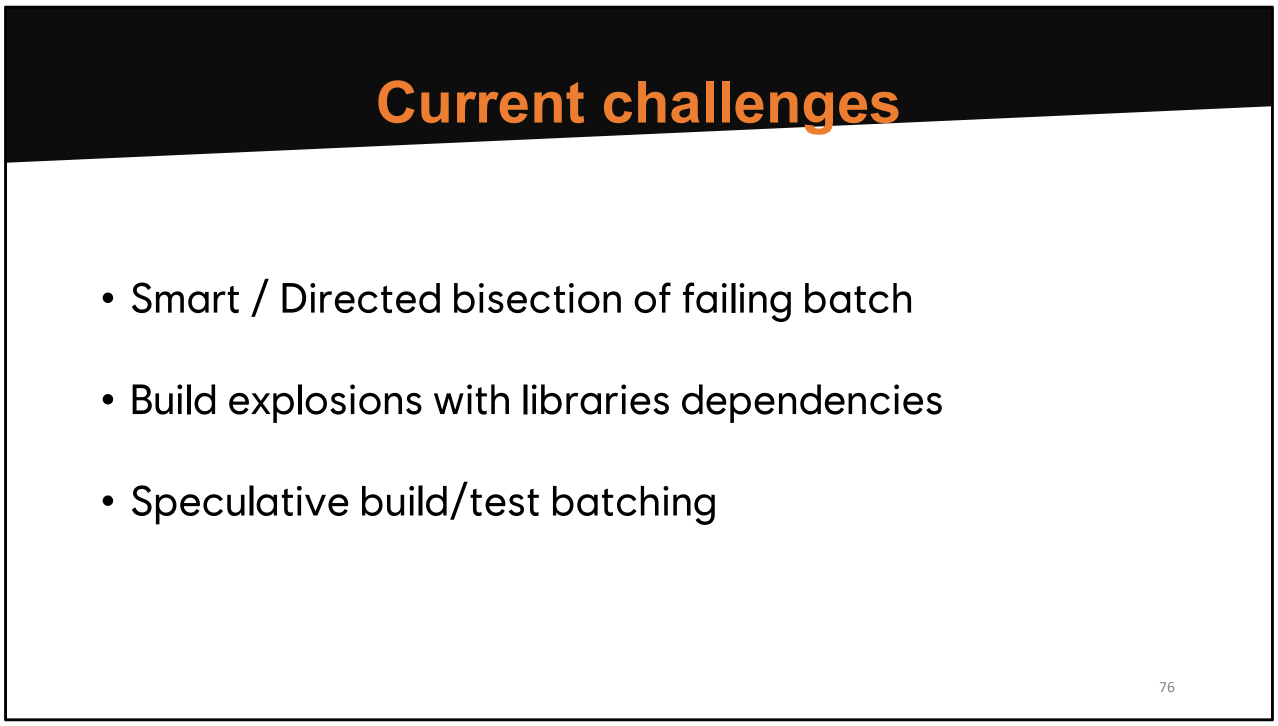


TESTS ROI

- #bugs introduced in covered code reduces ROI
- Lower ROI tests are to be scheduled last or ignored
- Very low ROI tests are to be investigated / removed / reworked

75

Other ways to orchestrate the tests is based on their ROI. For us the ROI of a test is simply the 1- the bug-introduction rate – or the amount of bug introduced over the amount of contributions – of a code-region covered by the test. If the bug introduction rate is 0.8 or 80%, then the ROI of the test is 0.2.



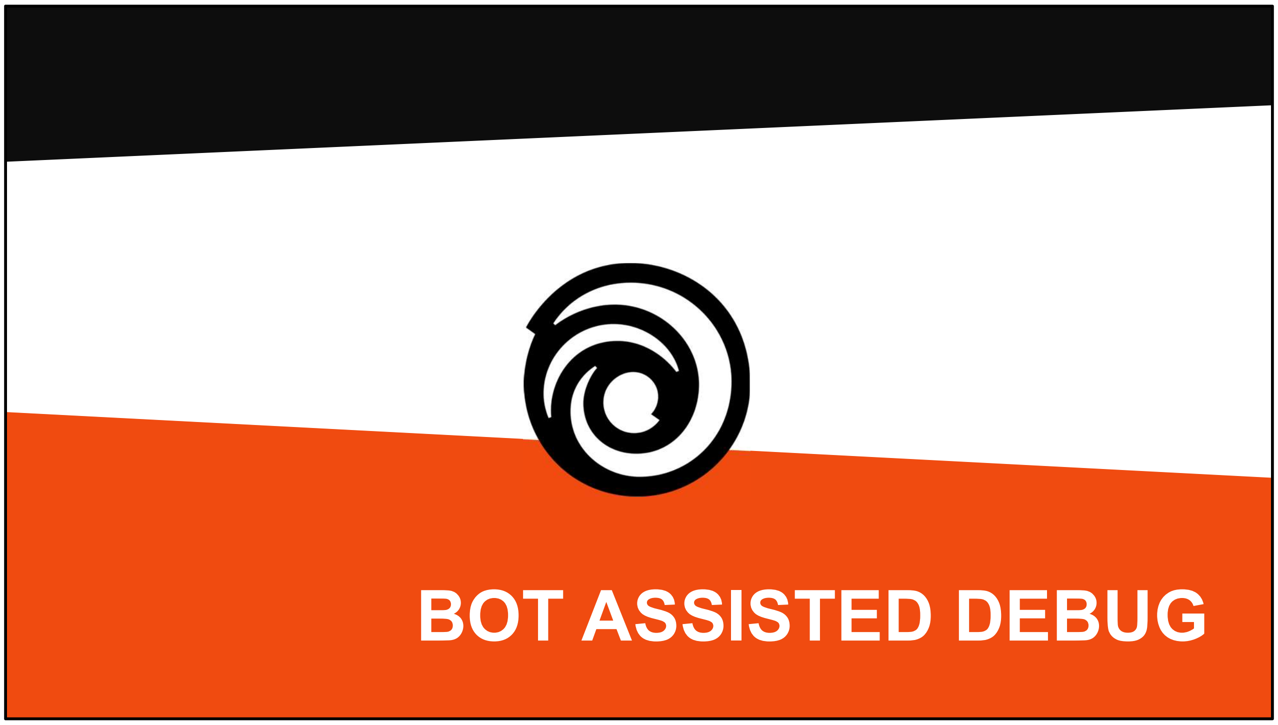
Current challenges

- Smart / Directed bisection of failing batch
- Build explosions with libraries dependencies
- Speculative build/test batching

76

The challenges we currently face on CI are mainly linked to the volume of contributions and the time it takes to build and run the test suite of a AAA game. We face with an explosion of build as we embrace cicd principles at all the levels – from the low level mathematical library to the game itself – and we are force to batch contributions in the same cicd pipeline. When a pipeline composed of dozens or more contributions fails, investigation which contribution is the guilty one is often tricky. We can rely upon classical bisection, but this requires yet another set of builds.

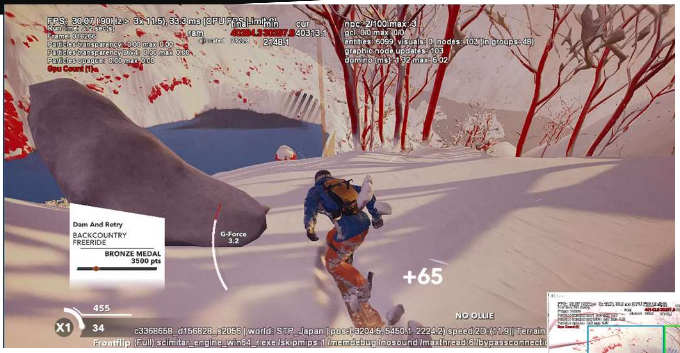
We are also interested in optimizing cicd pipelines by using speculative batching where an automated process would select which set of libraries to build/test in addition to a set of game-code changes to lower the number of builds while keeping a high level of confidence. This is specifically interesting for us as libraries and game-code co-evolve at the same time and at a high velocity.

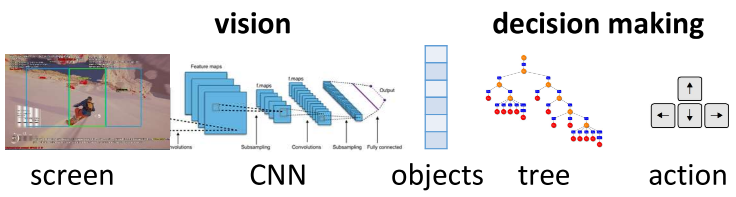


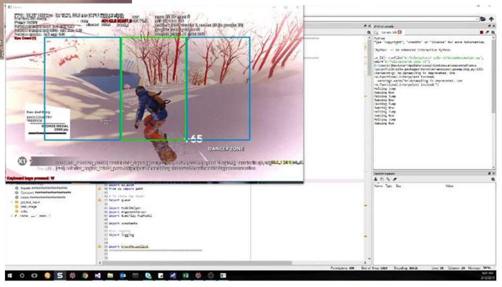
BOT ASSISTED DEBUG

Another tool we have to debug game is to train bot to play the game for us and report their findings.

PIXEL BASED-LEARNING





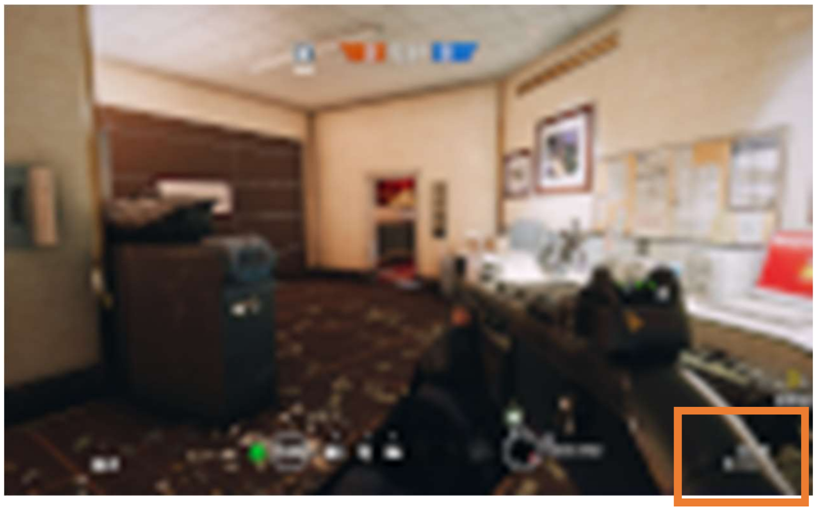


- And I want to illustrate them through the lens of ongoing work by my colleagues at Ubisoft Pune, who are building a bot for automated tests on Steep
- Here you can see the agent playing in the top left, along with its input in the bottom right, where you can see that it's running at a lower framerate and only analyzing the part of the screen within the 3 rectangles, because computations would be too heavy otherwise
- I first want you to notice that there are a lot of red trees and red rocks on the screen: this is because everything is covered in snow in Steep, which made it very difficult to identify obstacles with classical computer vision algorithms
- So it was decided to change the textures to make it easier to detect important objects on the screen
- In addition, here they decided to split the AI logic in two parts: first train an object detection algorithm, which is the vision part of the AI
- Then use a decision making module that takes the detected objects as input, instead of working directly from the game screen
- And one major reason to do so is to save computations, because when you train your AI directly from pixels, it has to learn both a vision module *and* a decision making module at the same time

// Video in the blog-post at laforge.ubisoft.com

WHT NOT LEARN FROM PIXEL

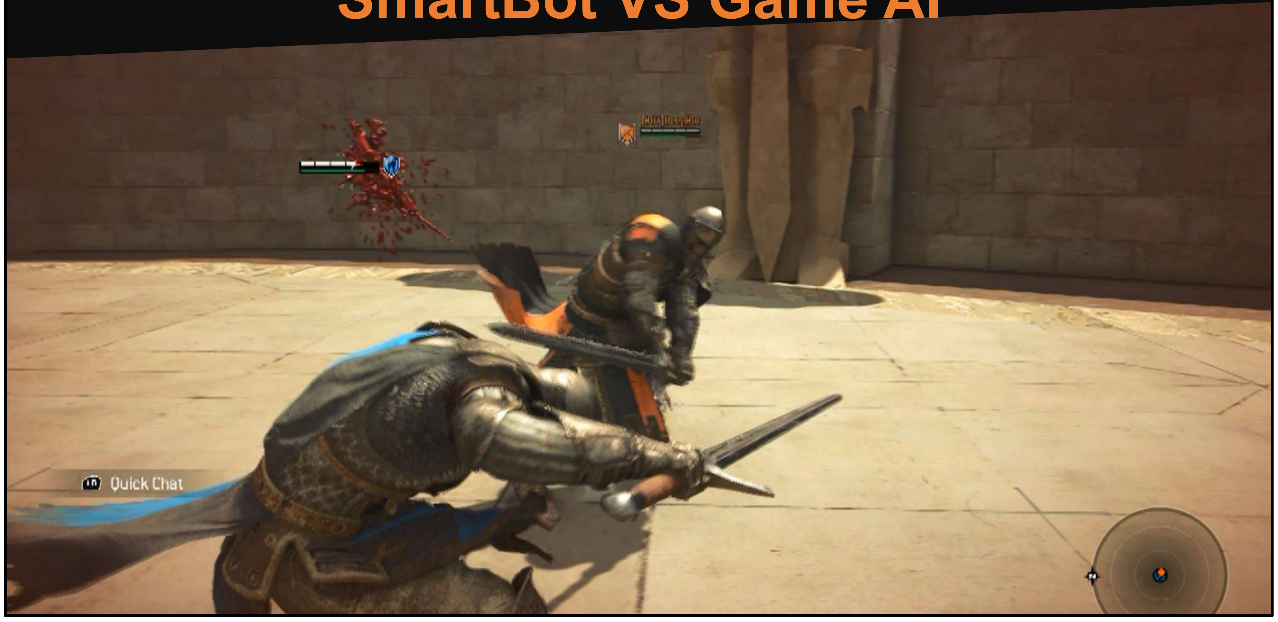
- Complex training
- Large neural network
- Costly GPU rendering
- Partial observability
- **Less than ideal for…**
  - **UI details**
  - **Sound**
  - **Reward**



The problem with pixel learning is that is extremely complex as it requires a lot of GPU power. The game is rendered on the GPU and the training is done the GPU too. Also, it does not tell the full story, the game states that could be useful to learn from could be hidden or be very small details in the UI.

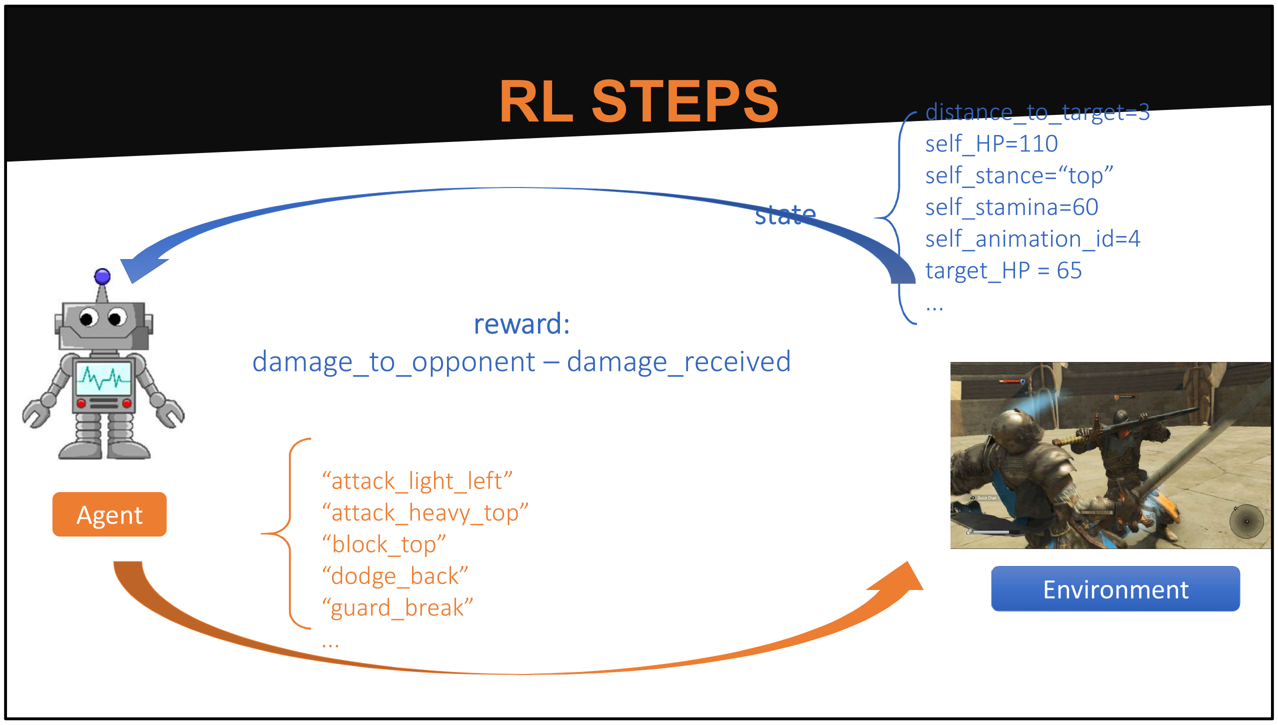
// Video in the blog-post at laforge.ubisoft.com

SmartBot VS Game AI



- And while we were experimenting with a reinforcement learning prototype on For Honor, we happened to find a weakness in the AI
- Here you can see in orange our reinforcement learning bot (let’s call it the « SmartBot »), that was trained to fight against an existing game AI in blue
- And what you will notice is that what the SmartBot is stepping back, and by doing so it forces the game AI to sprint forward
- And what happens is that the game AI was vulnerable during sprinting, and the SmartBot used this weakness to safely punish it
- So after seeing this kind of result, we decided to build a fully automated test pipeline to help discover potential weaknesses or exploits in the game AI

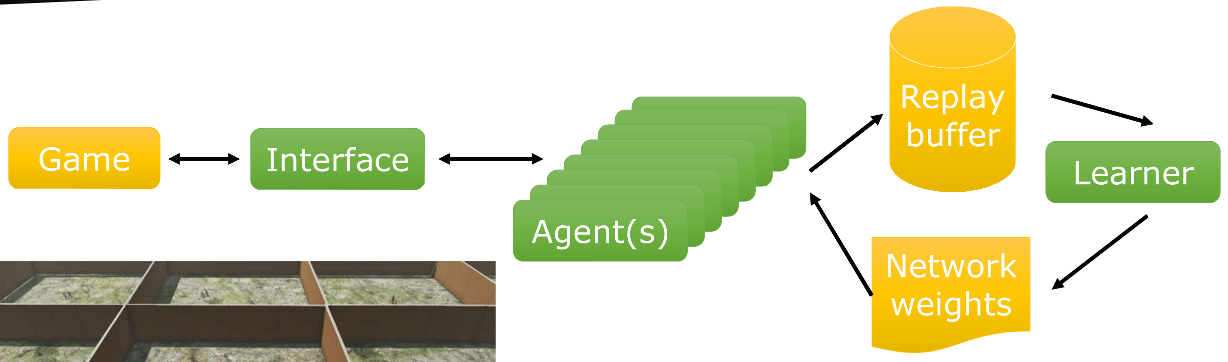
// Video in the blog-post at laforge.ubisoft.com

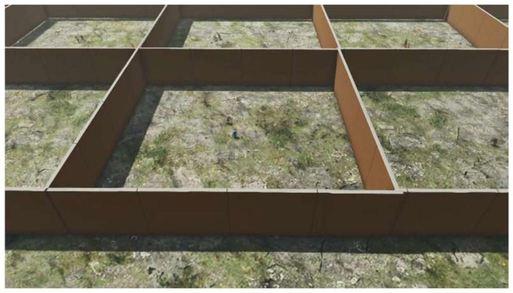


RL STEPS

state
distance_to_target=3
self_HP=110
self_stance="top"
self_stamina=60
self_animation_id=4
target_HP = 65
…
reward:
damage_to_opponent – damage_received
Agent
"attack_light_left"
"attack_heavy_top"
"block_top"
"dodge_back"
"guard_break"
…
Environment

DQN with distributed experience replay

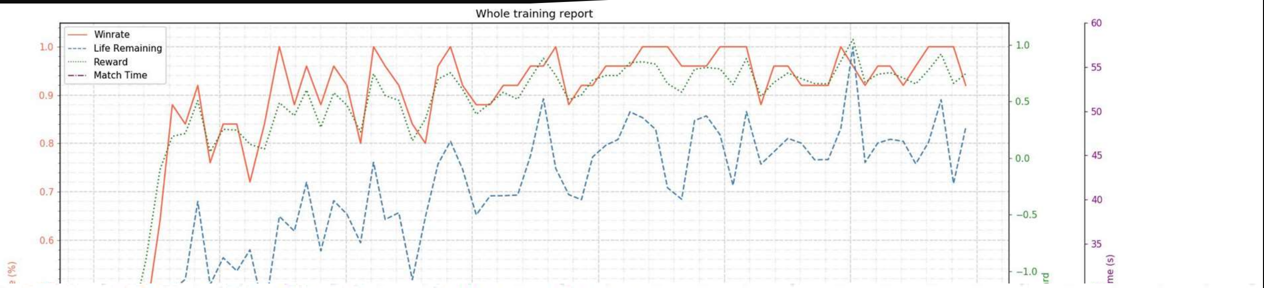




Distributed Prioritized Experience Replay
(Horgan et al. 2018)

- When we implemented this reinforcement learning loop and started training our agent with the Deep Q-Network algorithm, we quickly realized the need to collect data at a faster rate in order to speed up training
- To this end, we created this special map that is divided in 10 cells, each of which hosting a unique 1v1 match
- By having multiple agents fighting in the same game instance, we can collect data much more quickly without having to run the game multiple times
- In addition, here we were able to run the game at twice real-time to collect data even faster
- The architecture we used is pretty close to the one you can find in a paper called “Distributed Prioritized Experience Replay”
- We used one Python process per agent (here each red box is a unique Python process)
- We have a single process communicating with the game, so as to minimize the communication overhead
- All agents dump the data they collect in the same database, called the “replay buffer”, which is used by another process to learn the weights of the neural network
- This neural network is then fed back to the agents so that their behavior improves over time

Results



Action Info: Distribution of the actions

	1st	2nd	3rd
All actions distribution	Dodge_Back(23.5%)	Attack_AoE_Special_Strike_Part1(17.9%)	Dodg
% damage done	Attack_AoE_Special_Strike_Part1(68.4%)	Attack_AoE_Special_Strike_Part2_(16.0%)	Attac

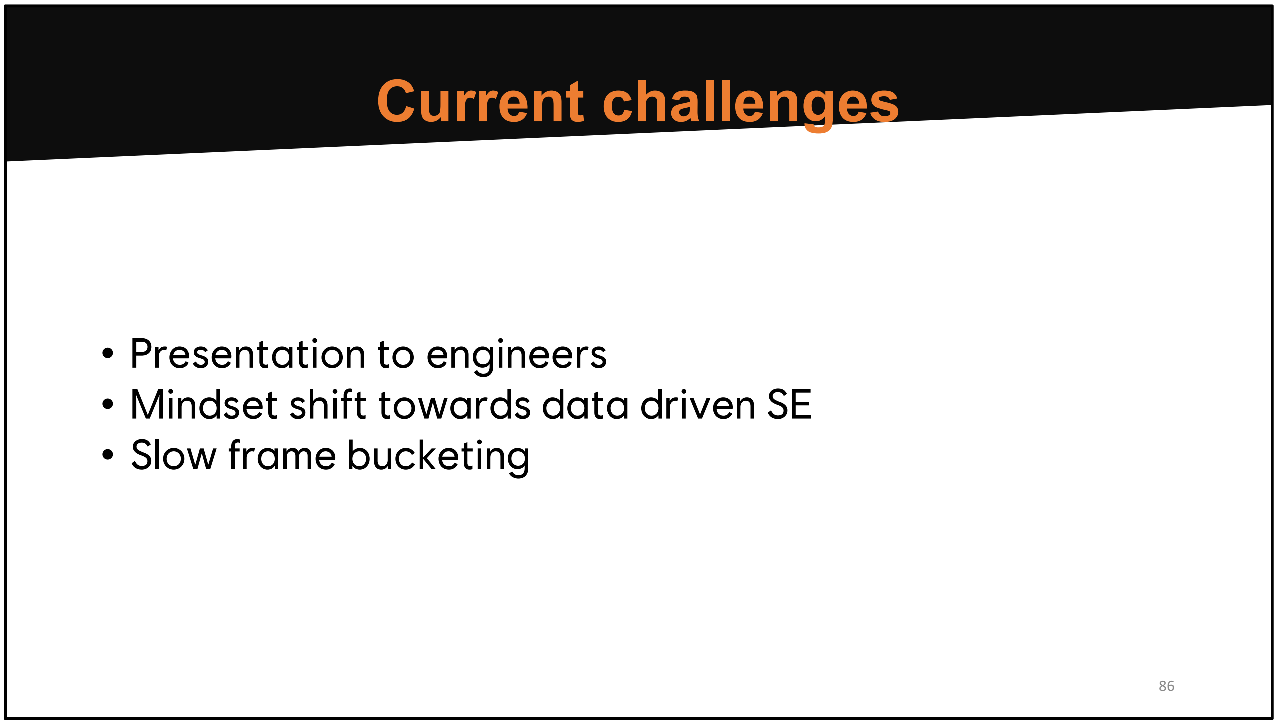
Using the findings of the bot we can debug and profile our games. For instances slow-frames that the bot triggers can be further analyzed and the actions that the bot is taking towards victory are recorded. Game designers can then look at the data is estimate if the game is still fun to play and balanced. For instance, a bot achieving a 100% win-rate against all enemy by exploiting a flaw in a new gameplay feature would be caught.

This allows us to reach new level of debugging where we are not only concerned by does it compiles? Does pass the tests? To Did I break the game? Are the performances still ok in gamer-like conditions?

RL CARS

We are also training cars to drive around with various behaviors and explore the driving centric games for us.

// Video in the blog-post at laforge.ubisoft.com



Current challenges

- Presentation to engineers
- Mindset shift towards data driven SE
- Slow frame bucketing

86

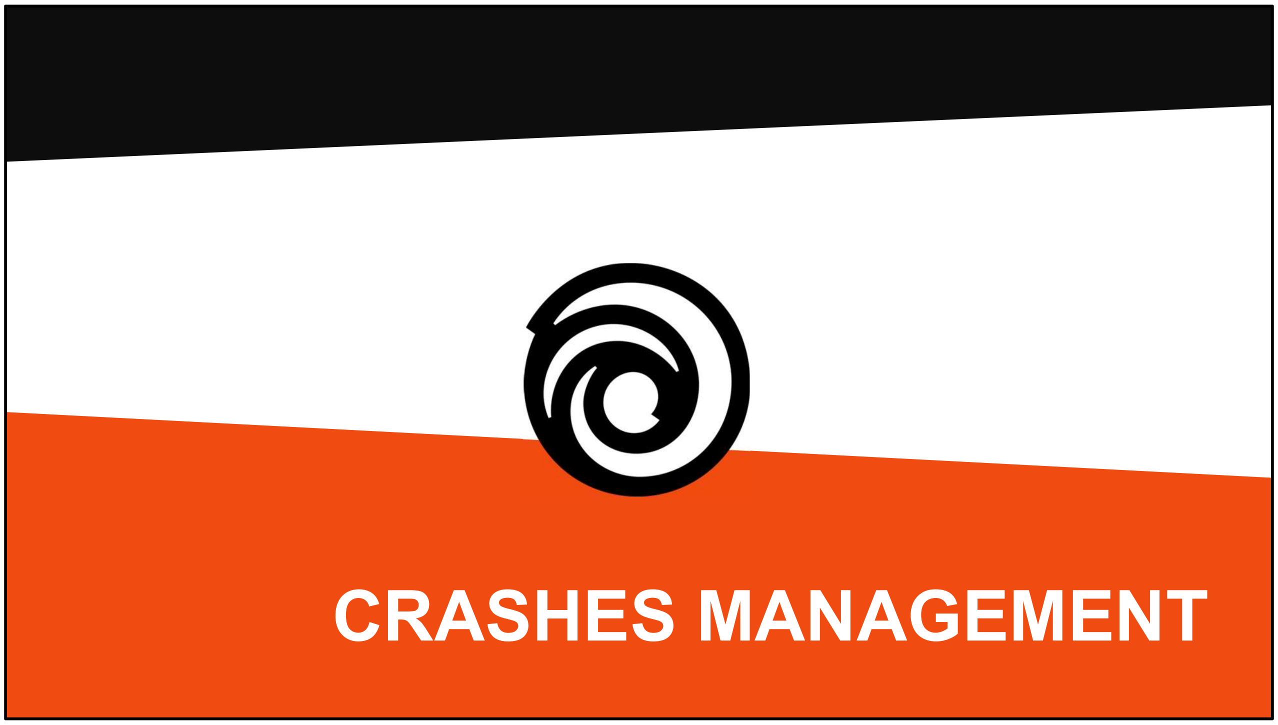
Now for the challenges, in this conference we heard a lot of good points regarding ML 4 SE and SE 4 ML where we advocate for knowledge transfers to be able to better apply machine learning technique on software engineering activities and leverage decades of software engineering research while building machine learning approaches.

To add to the mix of challenges, I'd argue that we also need a mindset change of software-engineers; perhaps taught in colleges, to be able to leverage data-analytics when developing software. I believe that ML techniques advanced enough to propose patches to software engineers without false positive would be used in the wild.

However, we are building complex models too fast without a proper understanding of the underlying data and how it was produced or generated. One of the telltales is building a complex deep-learning network that takes days to be trained only to be told by the practitioners: "I know".

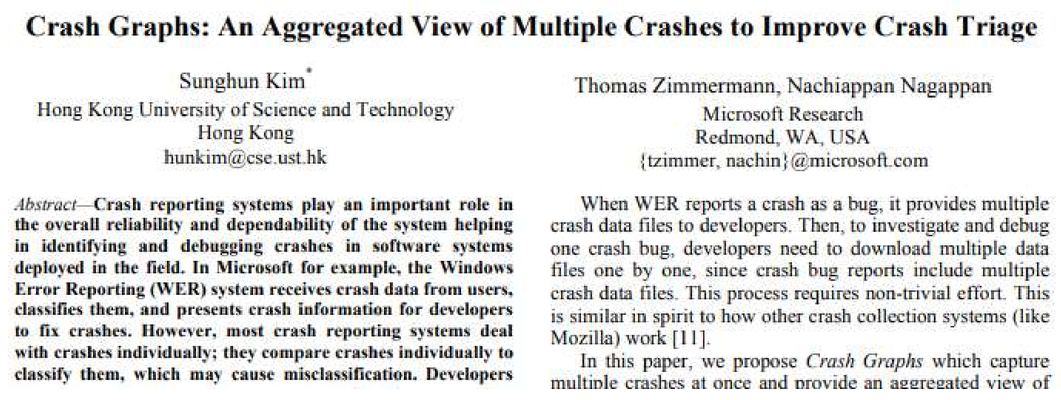
Well, if you knew, why is that bug still in the code? Could we have excluded it from the data? Our complex models are generalizing on facts we already know and avoiding this requires a mindset change.

It requires a mindset change from software practitioners because the data produced is likely to be used to train models and for now, the raw data is rigged with normal alerts, known bug and performances problems.



CRASHES MANAGEMENT

MD5 + CRASH GRAPH


Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage

Sunghun Kim*
Hong Kong University of Science and Technology
Hong Kong
hunkim@cse.ust.hk

Thomas Zimmermann, Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
{tzimmer, nachin}@microsoft.com

Abstract—**Crash reporting systems play an important role in the overall reliability and dependability of the system helping in identifying and debugging crashes in software systems deployed in the field. In Microsoft for example, the Windows Error Reporting (WER) system receives crash data from users, classifies them, and presents crash information for developers to fix crashes. However, most crash reporting systems deal with crashes individually; they compare crashes individually to classify them, which may cause misclassification. Developers**

When WER reports a crash as a bug, it provides multiple crash data files to developers. Then, to investigate and debug one crash bug, developers need to download multiple data files one by one, since crash bug reports include multiple crash data files. This process requires non-trivial effort. This is similar in spirit to how other crash collection systems (like Mozilla) work [11].

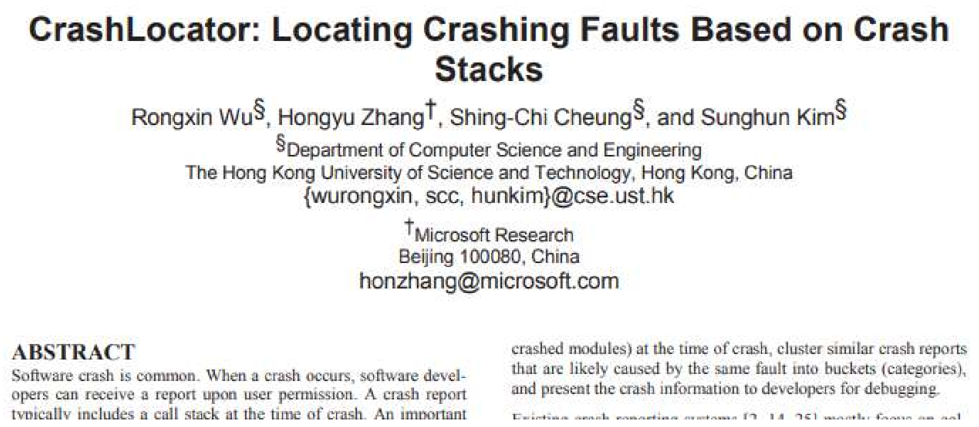
In this paper, we propose *Crash Graphs* which capture multiple crashes at once and provide an aggregated view of

88

Despites all our efforts in automated debugging and testing, the game still crashes from time to time. To handle these crashes and bucket them to be assign, we are using an MD5 approach + an adapted version of crashgraphs.

In the MD5 approach we simply MD5 the last few lines of the crash stack while removing frames that are known to be platform specific to create a specific signature. Crash graph is a bit costlier to run but identify crashes that are very close to each other's and should be on the same bucket. We are using this when the unique MD5 signature does not match any known bucket.

Faults Locations



CrashLocator: Locating Crashing Faults Based on Crash Stacks

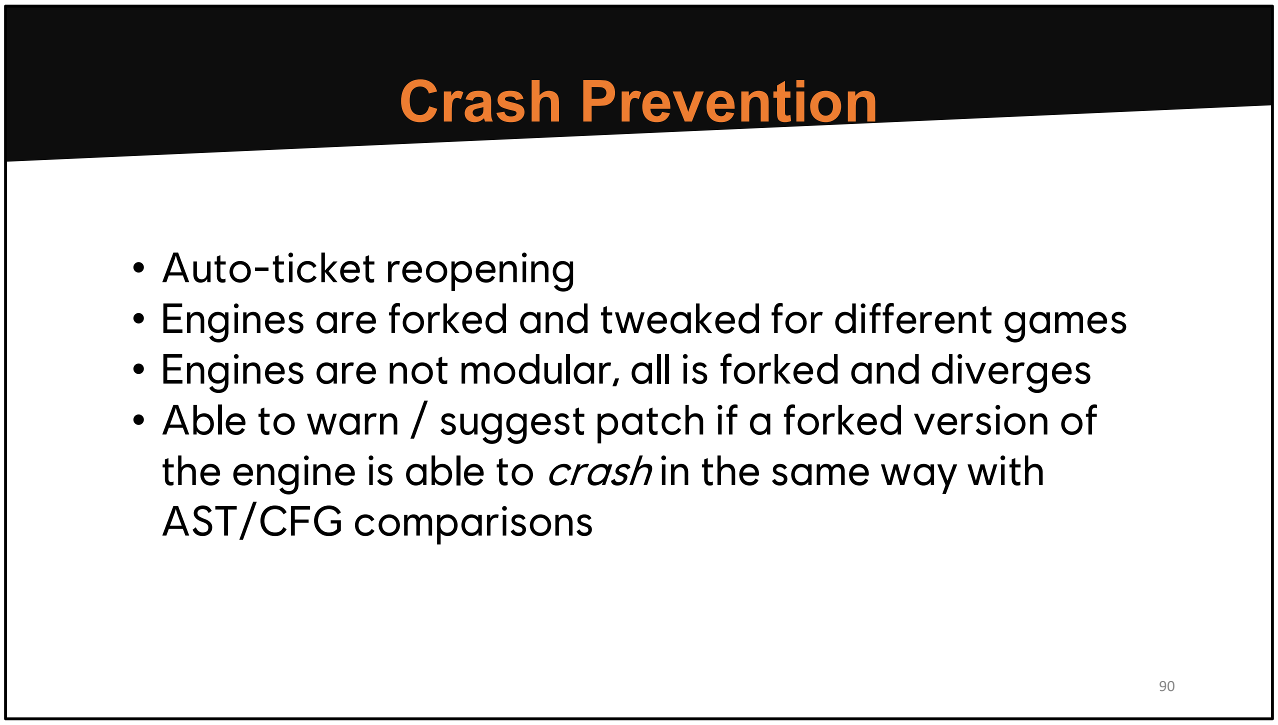
Rongxin Wu[§], Hongyu Zhang[†], Shing-Chi Cheung[§], and Sunghun Kim[§]
[§]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{wurongxin, scc, hunkim}@cse.ust.hk
[†]Microsoft Research
Beijing 100080, China
honzhang@microsoft.com

ABSTRACT
Software crash is common. When a crash occurs, software developers can receive a report upon user permission. A crash report typically includes a call stack at the time of crash. An important

crashed modules) at the time of crash, cluster similar crash reports that are likely caused by the same fault into buckets (categories), and present the crash information to developers for debugging.

Existing crash reporting systems [2, 14, 25] mostly focus on col

We are also using known approaches to locate the faults based on the crash stacks.

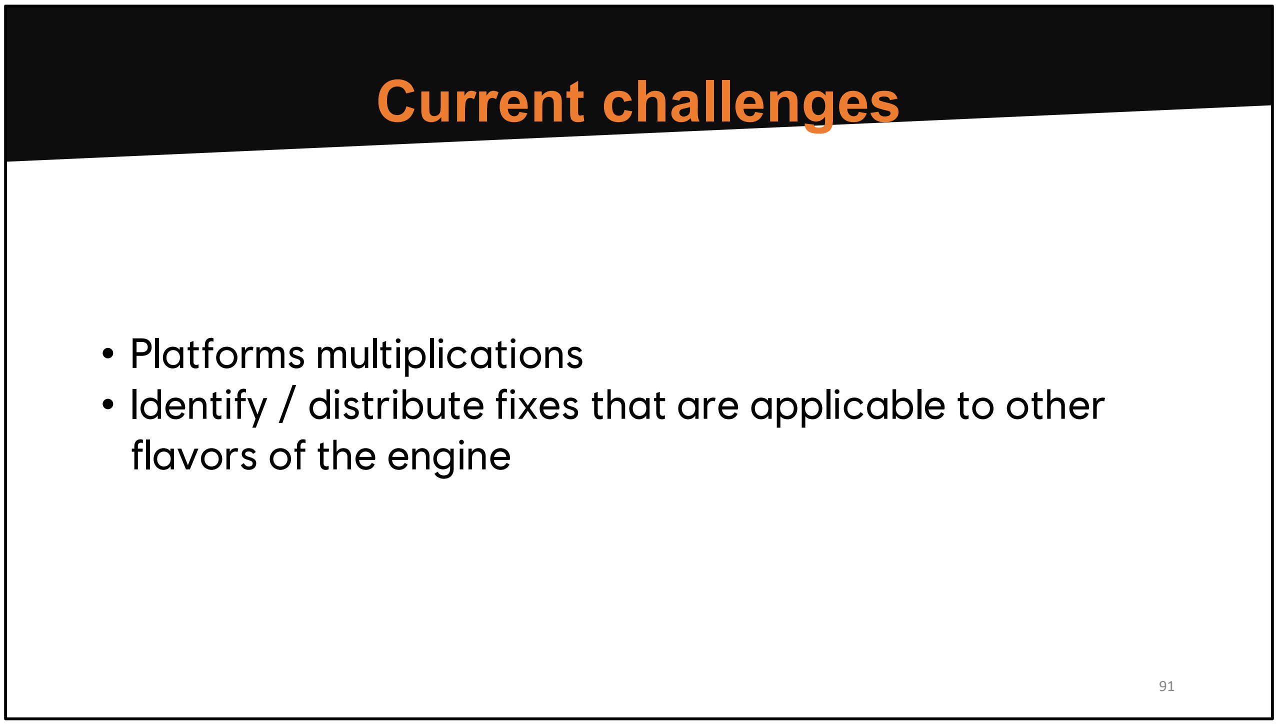


Crash Prevention

- Auto-ticket reopening
- Engines are forked and tweaked for different games
- Engines are not modular, all is forked and diverges
- Able to warn / suggest patch if a forked version of the engine is able to *crash* in the same way with AST/CFG comparisons

90

To prevent crash from occurring or re-occurring, we must handle the diversity of our game engines. Game engines are not modular piece of software and, while the core stays stable, the upper layers and often branched and tweaks for a game specific intent. We are working on identifying, using the AST and CFG of a fault identified by the crash stack on one game if other games could crash the same way.



Current challenges

- Platforms multiplications
- Identify / distribute fixes that are applicable to other flavors of the engine

91

The challenges related to the crash management are mainly due to the multiplications of platforms that could crash as we have to create and maintain code for each and every one of them. Decoding the callstack is, of course, not standard across platforms and specifications not always explicit. We are still investigating the perfect way to handle crashes when confronted with so many platforms, games and players in order to create buckets of crash that are automatically triaged and easily addressable by dev teams.

We also looking into ways of tackling our internal code-divergence and how to detect and apply fixes from hard forks into the original branch.

Where to learn more

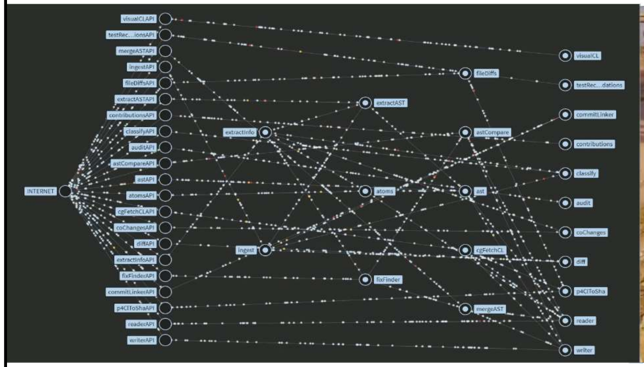
- SIGGRAPH'18-19
- SEMLA'19
- GDC'19
- CPPCON'18
- MSR'18

- Come find me today

- ASE'20 ? ;)

92

HIRING & NEW COLLABS



https://github.com/MathieuNls/clever-challenge

94

I also take the opportunity to do some self-promotion and tell you that we are hiring exceptional candidates because, I can see on the right, one of our character is swimming on land, so there's still work to be done. We are looking for software engineers with data science experience to refines our models.

We are also aggressively looking for software engineers that can help us operate all of these models and approaches in a micro-services environment and build the future of automated debug & profiling at scale. We pay for you to relocate in Montreal, Canada and you get to play with what I presented today applied to un-released titles.

Techs:
- AST, CFG, Patterns, Smells, SW Metrics, ML, RL
- Python, Go, C#, C++, R, WPF, Vue
- Docker, K8s, DevOPS, Git, Perforce
- Redis, Cassanda, Mysql, Nginx
- Scikit Learn, Tensorflow, NLP

We are also looking for trainees from academia that what to test their ideas on real-dataset and professors to build new long-lasting collaborations.

// Video in the blog-post at laforge.ubisoft.com

UBISOFT

LA FORGE

TECHNOLOGY GROUP
UBISOFT

LAFORGE.UBISOFT.COM
@MATHIEUNLS

More information about laforge at lagorge.ubisoft.com and ubisoft at montreal.ubisoft.com/en/.

You can find me on twitter @mathieunls where I tweet about our work and software engineering research in general.