



Mining Development Data to Understand and Improve Software Engineering Processes in HPC Projects



Boyana Norris
University of Oregon

norris@cs.uoregon.edu



Motivation

- Software development productivity is affected by many factors, many of which are difficult or impossible to measure (accurately).
- Here, we wish to focus on the things we **can** measure and provide tools that can help in understanding how existing or new software practices may affect these observables.
- Tools are meant to be used -- so we will show some simple ways in which *anyone* can begin exploring their own project's data without significant up-front investment of time and effort.

Murphy-Hill, E., Ciera Jaspan, Caitlin Sadowski, D. Shepherd, Michael Phillips, C. Winter, Andrea Knight, Edward K. Smith and M. Jorde. "What Predicts Software Developers' Productivity?" *IEEE Transactions on Software Engineering* 47 (2021): 582-594.

Motivation

(Am I just making things up?)

- Top three (on average):**
- Job enthusiasm (F1)
 - Peer support for new ideas (F2)
 - Useful feedback about job performance (F11)

	Google (n=407)		ABB (n=137)		NI (n=78)		estimate μ (r)	Google Analysts (n=88)		
	estimate	error	estimate	error	estimate	error		estimate	error	diff
I am enthusiastic about my job F1	0.414 *	0.049	0.386 *	0.090	0.484 *	0.089	0.43 (0.051)	0.422	0.097	-0.01
People on my project are supportive of new ideas F2	0.337 *	0.059	0.318 *	0.090	0.312 *	0.102	0.32 (0.013)	0.461	0.133	-0.12
My job allows me to make decisions about what methods I use to complete my work F3	0.298 *	0.058	0.380 *	0.093	0.237	0.106	0.30 (0.061)	0.365	0.157	-0.07
My job allows me to make my own decisions about managing my time F4	0.293 *	0.066	0.316 *	0.078	0.237	0.121	0.28 (0.042)	0.487	0.106	-0.19
People who manage my project are highly capable, efficient, thorough, communicative, and cooperative F5	0.318 *	0.053	0.241 *	0.084	0.264	0.112	0.27 (0.04)	0.287	0.105	0.03
The information supplied to me (bug reports, user stories, etc.) is accurate F6	0.233 *	0.061	0.161	0.087	0.418 *	0.111	0.27 (0.132)	-	-	-
I feel positively about other people on my project F7	0.291 *	0.063	0.240	0.103	0.278	0.148	0.27 (0.027)	0.611	0.135	-0.32
My job allows me to use my personal judgment in carrying out my work F8	0.372 *	0.058	0.242	0.090	0.172	0.100	0.26 (0.101)	0.519	0.127	-0.15
My project resolves conflicts quickly F9	0.295 *	0.048	0.207	0.085	0.272	0.115	0.26 (0.046)	0.456	0.111	-0.16
People who write code for my software are highly capable, efficient, thorough, communicative, and cooperative F10	0.348 *	0.058	0.177	0.084	0.245	0.124	0.26 (0.086)	-	-	-
I receive useful feedback about my job performance F11	0.245 *	0.050	0.262	0.076	0.259 *	0.091	0.26 (0.009)	0.220	0.125	0.02
My job requires me to use a number of complex or high-level skills F12	0.304 *	0.055	0.246	0.095	0.193	0.086	0.25 (0.056)	0.286	0.130	0.02
My job involves a great deal of task variety F13	0.163 *	0.057	0.235 *	0.069	0.336	0.133	0.24 (0.077)	-0.010	0.128	0.17
People who work on my software's requirements and design are highly capable, efficient, thorough, communicative, and cooperative F14	0.289 *	0.050	0.174	0.076	0.267 *	0.094	0.24 (0.061)	-	-	-
I use the best tools and practices to develop my software F15	0.445 *	0.052	0.190	0.052	0.056	0.109	0.24 (0.191)	0.501	0.144	-0.06
Knowledge flows adequately between the key persons in our project F16	0.251 *	0.048	0.222 *	0.080	0.198	0.106	0.22 (0.026)	0.256	0.106	-0.01

I use the best tools and practices to develop my software F15 0.445 * [Bar chart]

Top factor at Google!

Fig. 4: 48 factors' correlation with developers' and analysts' self-rated productivity at three companies.

Other software-related factors

Murphy-Hill, E., Ciera Jaspán, Caitlin Sadowski, D. Shepherd, Michael Phillips, C. Winter, Andrea Knight, Edward K. Smith and M. Jorde. "What Predicts Software Developers' Productivity?" *IEEE Transactions on Software Engineering* 47 (2021): 582-594.

A few other positively rated software-related factors (in decreasing order of average scores):

- + My project's bug finding process is efficient and effective.
- + The software process my project uses is well defined.
- + My software reuses code, such as by using APIs, rather than duplicating it.
- + My software's architecture mitigates risks (e.g., security vulnerability, changes in requirements, etc.).

Negative impact on productivity:

- My software requires extensive data storage.
- My software is extremely complex.
- My software's platform (e.g. development environment, software stack, hardware stack) changes rapidly.
- Extensive documentation is required to use my software at different points in its lifecycle.

	Google (n=407)		ABB (n=137)		NI (n=78)		estimate μ (σ)	■	■
	estimate	error	estimate	error	estimate	error			
My project's bug finding process is efficient and effective F20	0.294 *	0.047	0.092	0.076	0.217	0.100	0.20 (0.102)	■	■
The software process my project uses is well-defined F21	0.309 *	0.046	0.121	0.072	0.165	0.114	0.20 (0.098)	■	■
My software reuses code, such as by using APIs, rather than duplicating it F25	0.310 *	0.052	0.030	0.074	0.221	0.144	0.19 (0.143)	■	■
My software's architecture mitigates risks (e.g., security vulnerabilities, changes in requirements, etc.) F27	0.313 *	0.050	0.062	0.087	0.141	0.104	0.17 (0.128)	■	■

Murphy-Hill et al. study conclusions

“A notable outcome of the ranking is that the top 10 productivity factors are non-technical. This is somewhat surprising, given that most software engineering research tends to focus on technical aspects of software engineering, in our estimation.

Thus, a vigorous refocusing on human factors may yield substantially more industry impact for the software engineering research community. For instance, answering the following questions may be especially fruitful:

- What makes software developers **enthusiastic** about their job? What accounts for differences in levels of enthusiasm between developers? What interventions can increase enthusiasm? This work can extend existing work on developer happiness [24] and motivation [25].
- What kinds of new ideas are commonly expressed in software development practice? What actions influence developers’ feelings of support for those ideas? What interventions can increase **support for new ideas**, while maintaining current **commitments**?
- What kinds of job feedback do software engineers receive, and what makes it **useful**? What kinds of feedback is not useful? What **interventions** can increase the regularity and usefulness of feedback?” [emphases mine]

What we are trying to do

Motivation and enthusiasm are ephemeral things that cannot be quantified easily.

On the other hand, we have many artifacts and associated metadata that are related (if indirectly) to productivity.

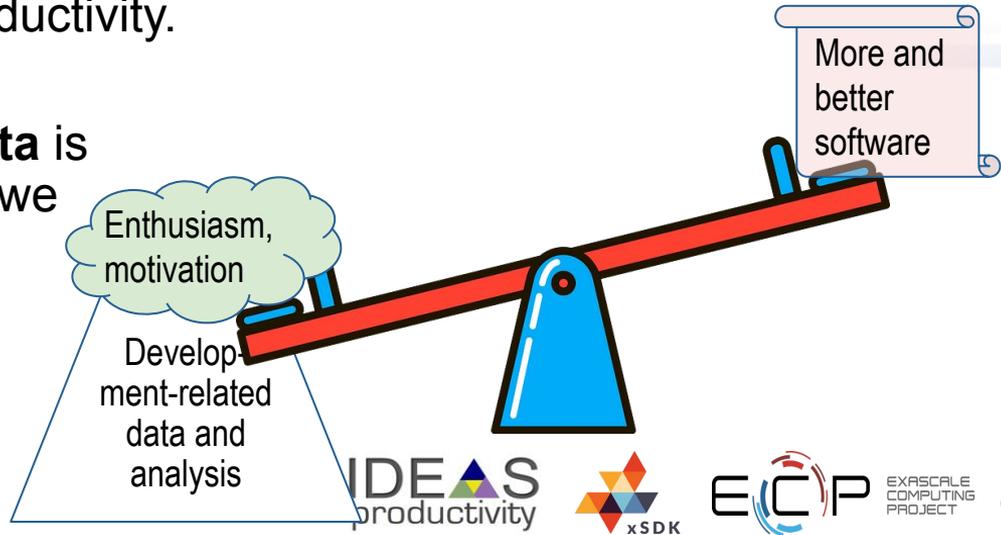
So, we are betting that using **some data** is better than not looking at any data, as we make decisions that are aimed at improving software development productivity and code quality.

This work is part of two DOE **ECP** projects:

IDEAS
productivity



IDEAS-ECP: foster and promote SE practices for better software
xSDK: Extreme-scale Scientific Software Development Kit



What this webinar is about

- We introduce a **flexible**, **efficient**, and **usable** software framework for **acquiring**, **storing**, **manipulating**, and **visualizing** development-related data.
- We demonstrate a few of its capabilities here; an ever growing number of patterns and other types of analysis are added continuously.
 - Contributions welcome! <https://github.com/HPCL/ideas-uo/>

ECP projects that may be present in examples in this webinar: Spack, LAMMPS, PETSc, Nek5000 E3SM, QMCPACK, QDPXX, LATTE NAMD, fast-export, Enzo, TAU2, xpress-apex, LATTE, NWChem, FLASH, Gingko.

Part I: Mining development metadata

Part II: Analyzing code

Part I: Mining your development data

Development data:

- Git metadata: commits, forks, branches, developers
- Issues and associated discussions
- Pull requests (github, gitlab) and associated discussions
- Mailing list archives

Goal: Analyze available data to help *formulate* and *answer* questions about development processes and their impact on productivity and code quality.

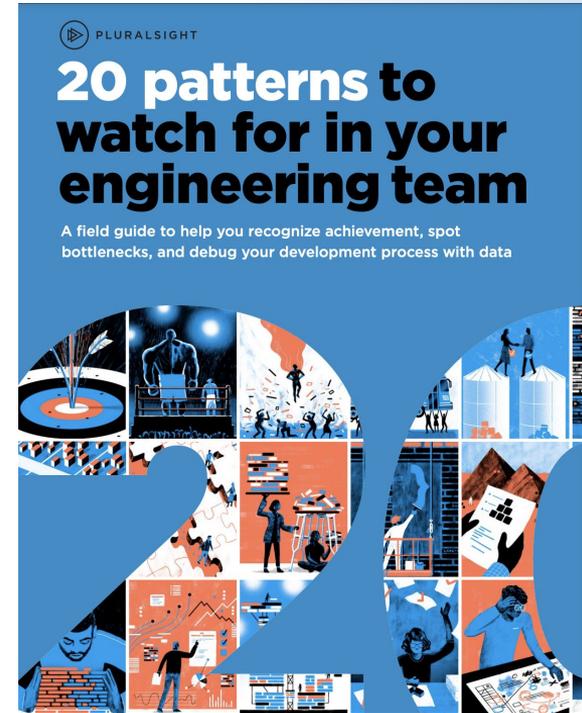
Questions that can be answered (in part) with IDEAS data analysis tools

- If I adopt practice **X**, how will metric **Y** be affected?
- How active is the developer community? (git, issues, PRs, emails)
- What parts of the code base could benefit from review or refactoring? (git, issues)
- What is the project's reliance on individual developers? (git)
- How are developers' contributions split among different categories? (git, manual labeling required)
- How engaged are the user and developer communities? (PRs, issues, mailing lists)
- What are some hot topics of discussion? (issues, mailing lists)
- How and on what do developers collaborate? (git, issues, mailing lists)
-

Common patterns with known implications

Pluralsight book (2019)¹:

“20 patterns is a collection of work patterns we’ve observed in working with hundreds of software teams. Our hope is that you’ll use this field guide to get a better feel for how the team works, and to recognize achievement, spot bottlenecks, and debug your development process with data.”

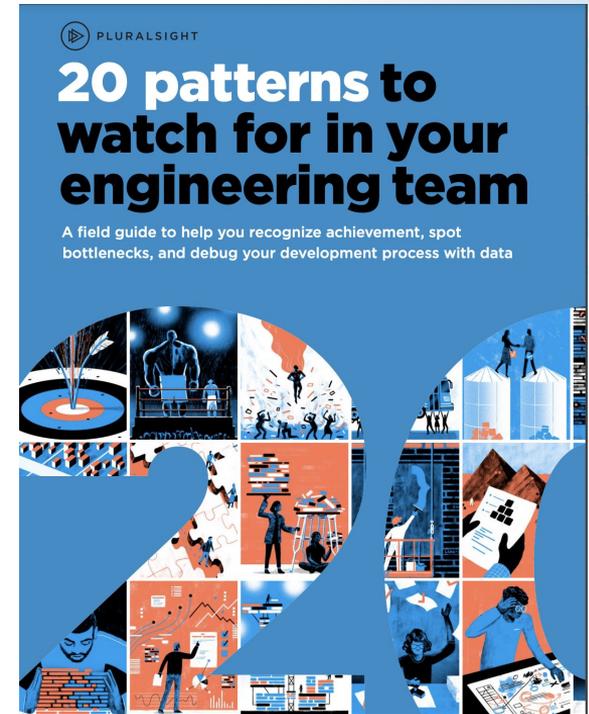


¹https://www.pluralsight.com/content/dam/pluralsight2/landing-pages/offers/flow/pdf/Pluralsight_20Patterns_ebook.pdf

Common patterns with known implications

Starting with these pattern descriptions, we can:

- Identify patterns that are relevant to HPC (open-source) software development.
- Characterize each pattern using data from revision control systems and developer communications.
- Inform decisions of the effects of adopting new SE practices.



¹https://www.pluralsight.com/content/dam/pluralsight2/landing-pages/offers/flow/pdf/Pluralsight_20Patterns_ebook.pdf

Example metrics

Metric	Description
Monthly bug fix rate	Computes cumulative count of bugs closed each month, starting from the beginning of the project. A higher value is not necessarily good, trends are more informative than individual values.
Monthly feature request rate	Number of new feature requests made by users. A higher value may indicate popularity/importance of the feature or missing functionality in the current software version.
Correlation of the number of issues with project age	Gives a good insight into the life cycle of the project by mapping the trend of issues raised over the lifetime of the project.
Commits, derived metrics	Characterizes some aspects of developer activity.
Number of issues	Project-related communications can be used to indicate community involvement and rates at which issues are resolved. For example, fast-growing projects can have significantly more activity in issues than more mature ones.
Issue categories	Identifies the types of issues that are reported in the repository. This information will be useful to derive correlation with other metrics.
Number of followers and watchers	Reports the code maturity and popularity. In addition, the number of followers can be correlated with the time it takes to fix reported issues.
Mailing list metrics	Relates to average time in discussion, most popular topic of interest, product activity based on type of emails, etc.
Number of contributors	Size of development community. When analyzed over time, we can estimate project turnover and identify different types of contributors.
Code complexity	Compute changes to code complexity metrics, e.g., cumulative size or cyclomatic complexity, over time.

Example pattern: Domain champion

The Domain Champion is an expert in a particular area of the codebase. They know nearly everything there is to know about their domain: every class, every method, every algorithm and pattern.

In truth, they probably wrote most of it, and in some cases rewrote the same sections of code multiple times.

The Domain Champion isn't just "the engineer who knows credit card processing"; it's all they ever work on. It's their whole day, every day.

Some degree of job specialization is essential and often motivating. But even within specialized roles there can be 'too much of one thing.' Managers must balance enabling a team member to unilaterally own the expertise, and encouraging breadth of experience.

How to recognize it

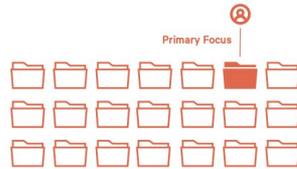
Domain Champions will always work in the same area of code. They'll also rewrite their code over and over, and you'll see it in churn and legacy refactoring metrics as they perfect it.

Domain Champions are deeply familiar with one particular domain. As a result, they'll typically submit their work in small, frequent commits and will show a sustained above average *Impact*.

Because no one else knows more than the Domain Champion, there's usually very little actionable feedback that

others can provide in the review process. As a result, Domain Champions will typically show low *Receptiveness* in incorporating feedback from reviews.

Domain Champions will seldom, if ever, appear blocked. Short-term, it's a highly productive pattern. But it's often not sustainable and can lead to stagnation, which of course can lead to attrition.



What to do

Assign tickets that focus on other areas of the codebase.

Of course, some engineers would prefer to stay where they are. It can be very enjoyable to do a task you're good at. And, it can be uncomfortable to take on work that requires information or skills you have less practice with. But effective managers will strive to challenge their team.

“The Domain Champion is an expert in a particular area of the codebase. They know nearly everything there is to know about their domain: every class, every method, every algorithm and pattern.”

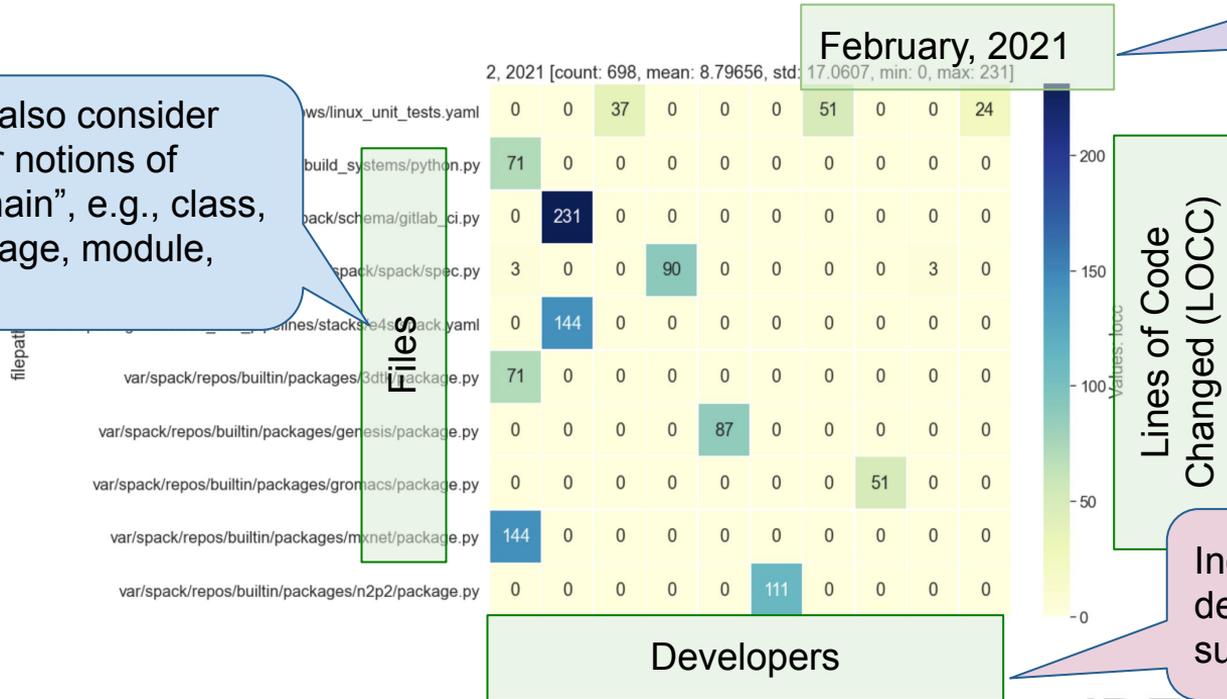
Why do we care?

- Can lead to great productivity
- Code quality implications
- Turnover effects on code -- what happens to the domain when the domain champion leaves?

Domain champion pattern: How do we detect it?

“The Domain Champion is an expert in a particular area of the codebase.”

Can also consider other notions of “domain”, e.g., class, package, module, etc.



Can be computed over any specific period.

Can consider different “change” metrics, more on that later.

Individual developers or subteams

Domain Champion Pattern: What, if anything, should we do?

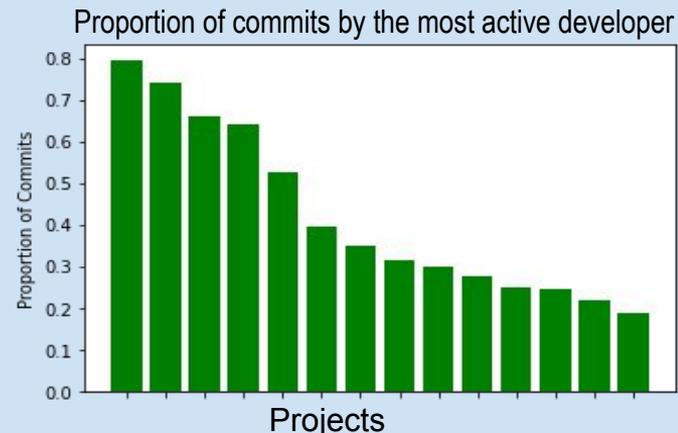
“The Domain Champion is an expert in a particular area of the codebase.”

- + Highly productive pattern in the short term.
- There's usually very little actionable feedback that others can provide in code reviews.
- Potentially not sustainable → can lead to stagnation.

😊 Possible actions:

- Assign the DC tickets for other areas of the code.
- Make an effort to involve others (e.g., new developers) in work in that domain.

One possible extension: Project Champion



Another example pattern: **Unusually high churn**

Churn is a natural and healthy part of the development process and varies from project to project. However, **Unusually High Churn** is often an early indicator that a team or a person may be struggling with an assignment.

In benchmarking the code contribution patterns of over 85,000 software engineers, Pluralsight's data science team identified that Code Churn levels frequently run between 13-30% of all code committed (i.e., 70-87% Efficiency), while a typical team can expect to operate in the neighborhood of 25% Code Churn (75% Efficiency).

Testing, reworking, and exploring various solutions is expected, and these levels will vary between people, types of projects, and stage in the software lifecycle. Given the variance, becoming familiar with your team's 'normal' levels is necessary to identify when something is off.

Unusually high churn levels aren't a problem in themselves. More likely, there are outside factors causing the problem.

An unusually high level of churn can be indicative of one of three behaviors:

- **Perfectionism:** When an engineers' standards of "good enough" are not aligned with the company's standard of "good enough." Engineers keep going back into the code to rewrite it because they think it can and should be better but may not add much to the actual functionality of the code.

- **They're struggling with the problem at hand.** This situation manifests differently than with Hoarding the Code (pattern #2), because in this case, the engineer initially thought they had correctly solved the problem, perhaps even sent it off for review, and then discovered it needed to be rewritten. Not just touched up. Rewritten.
- Or, most commonly, **issues concerning external stakeholders.** We see this with unclear or ambiguous specs, late arriving requirements, or mid-sprint updates to the deliverables.

How to recognize it

This pattern is characterized by **high levels of churn in the back of the sprint** or project. Watch for churn rates that climb significantly above the engineer's historical average (see the *Snapshot* and *Spot Check* reports), pairing that information with where they are in a project.



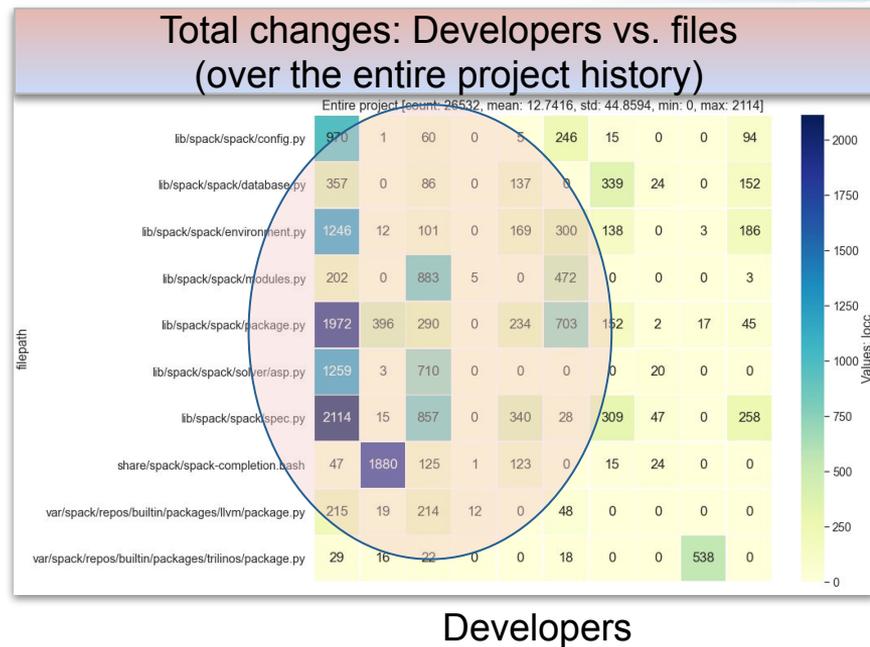
Unusually large amount of changes to single files or components may lead to development inefficiencies.

- + could be a sign of normal productive development
- may indicate need for more developer resources
- high conflict potential

20 patterns to watch for in your engineering team

Unusually high churn pattern: How do we detect it?

- Decide on the *granularity*. Some possibilities: modules, classes, files, functions.
- Define the *actors* -- groups of people (e.g., sub-teams), individual developers; doesn't have to be people, it can also be milestones or other project entities.
- Choose the *time period*.
- Choose the *churn* metric. Some examples: lines of code, cos (and other) difference between code versions, number of PRs, commits, number of files.

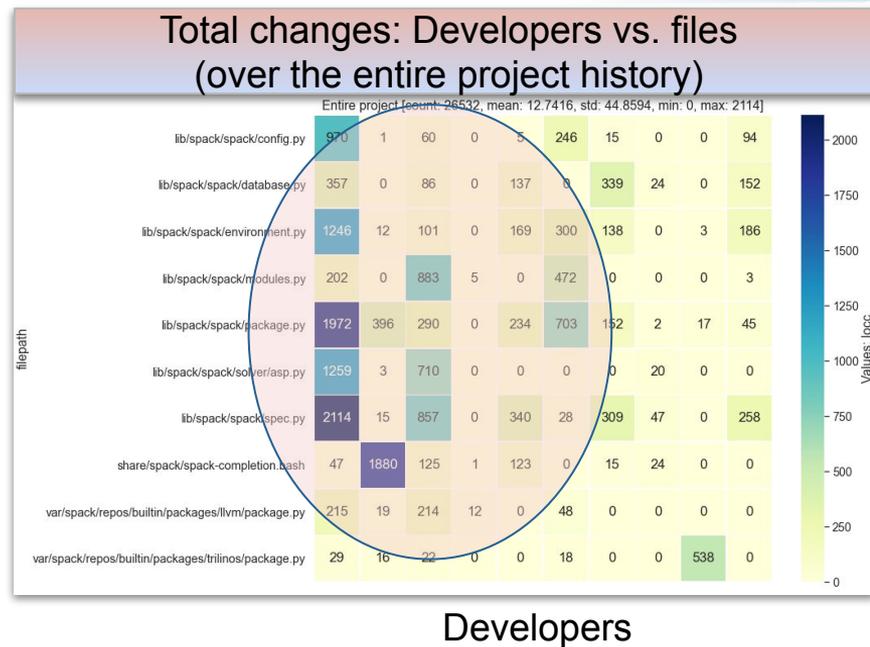


Unusually high churn pattern: What to do?

Unusually large amount of changes to single files or components may lead to development inefficiencies.

😊 Possible actions:

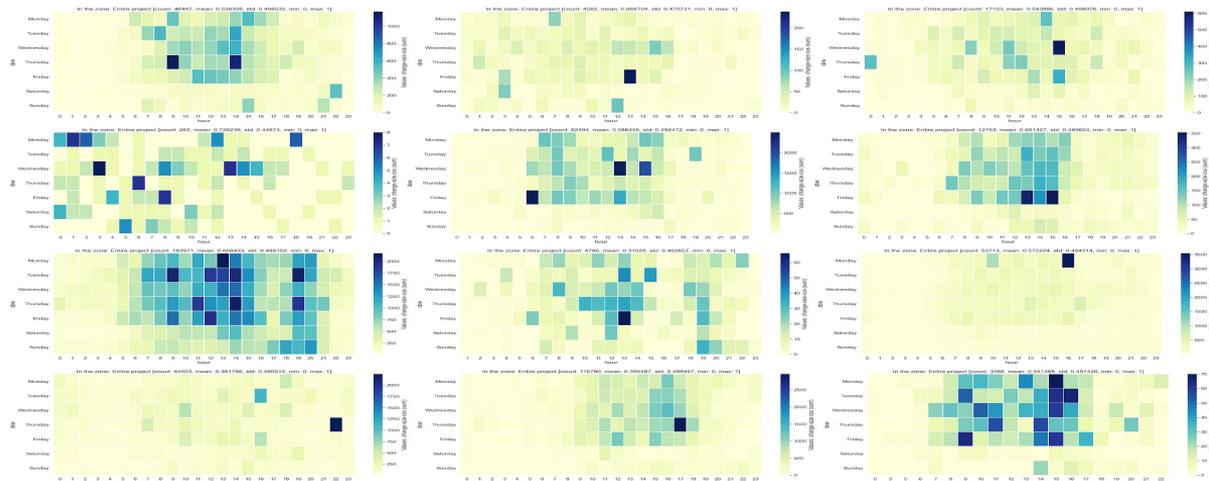
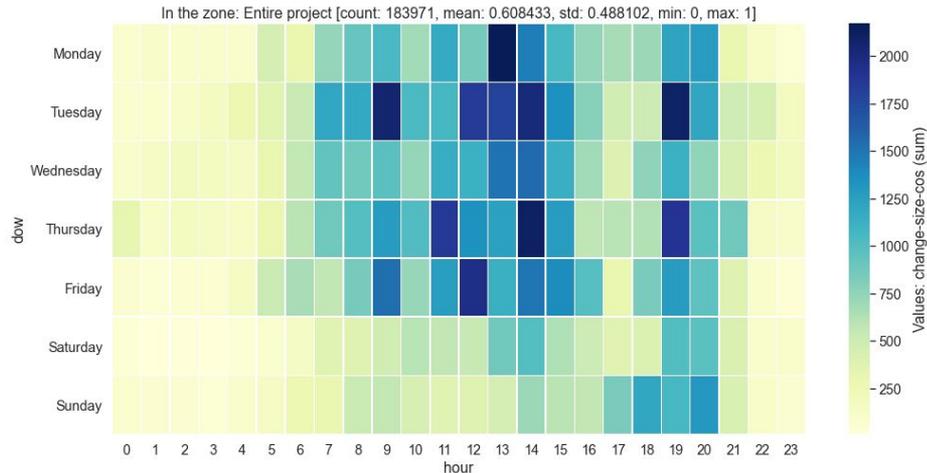
- Consider refactoring the high-churn project components
- Consider involving more developers in high-churn areas that are dominated by a single person



Pattern: In the zone

When are top contributors most productive?

- + Consistent high productivity during certain times of day
- Burnout, work-life balance



1. Choose *productivity* metric (LOCC, text difference, # commits, # PRs, etc.)
2. Choose *time period*
3. Choose visualization

In the zone pattern: What to do?

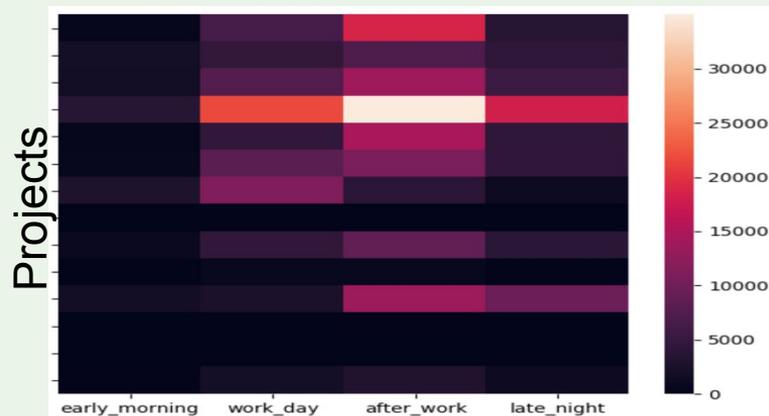
😊 Possible actions:

- Acknowledge the consistently high-performing developers.
- Recognize and acknowledge positive change in non-top developers (e.g., junior or new contributors).
- Consider the timing of and number of meetings during developers' most productive times.

When are top contributors most productive?

- + Consistent high productivity during certain times of day
- Burnout, work-life balance

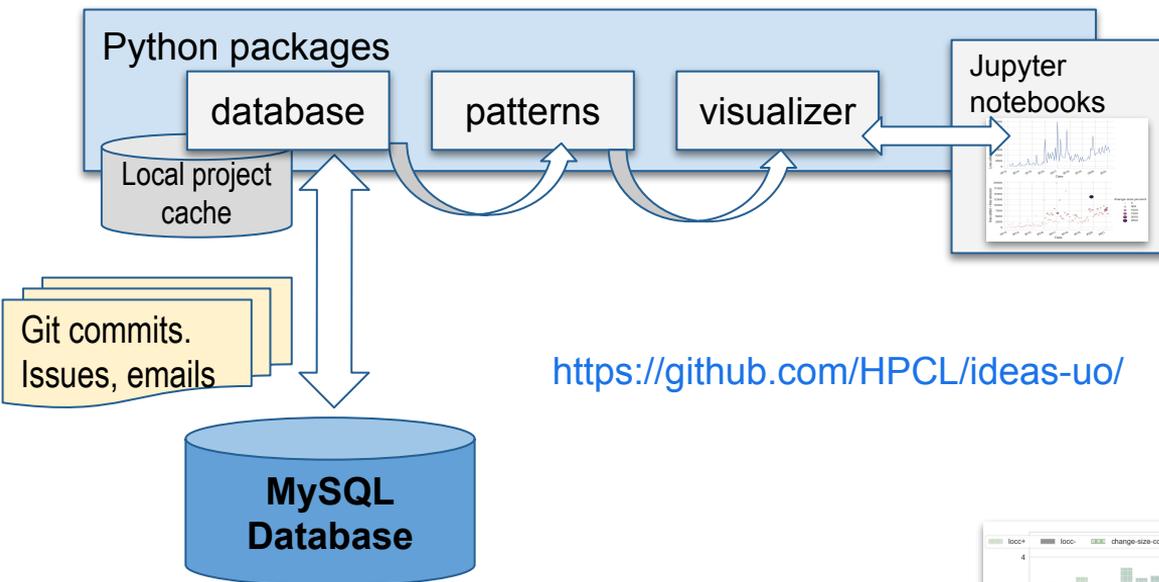
Bonus: Ability to quickly “see” what people are doing on average. Below: The average of over a dozen ECP projects' most active time periods.



Technical Details and Examples

<https://bit.ly/DevPatterns>

Implementation



<https://github.com/HPCL/ideas-uo/>

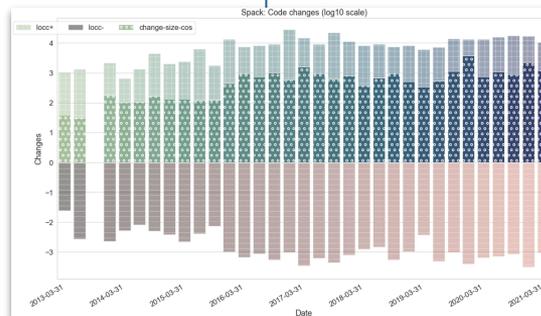
Example analysis workflow

```
from patterns.visualizer import  
Visualizer
```

```
vis =  
Visualizer(project_name='spack')
```

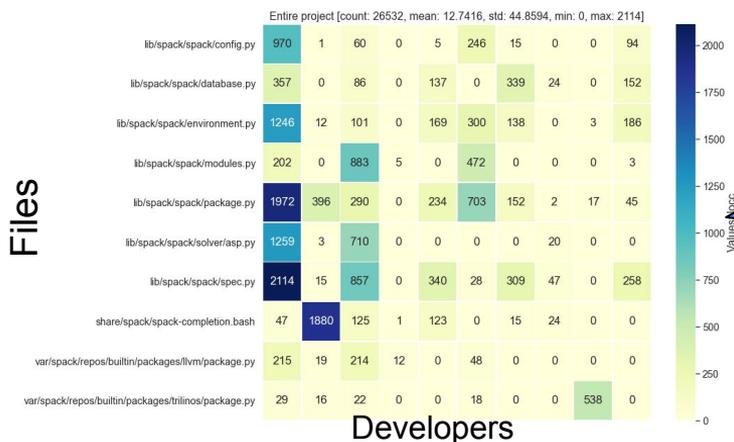
```
vis.get_data()  
INFO: Loaded local cached copy of  
spack data.  
INFO: Done computing averages. 64909  
commits (code only)
```

```
df =  
vis.plot_overall_project_locc(time_  
range=None, log=True)
```



Some currently available projects (more are being added constantly): LAMMPS, Spack, PETSc, Nek5000, NWChem, E3SM, QPMCPACK, qdpxx, NWChem, TAU,... Initial import can take up to 36 hours for some larger projects, but subsequent updates are fast and automated.

Impact of different “change” estimates

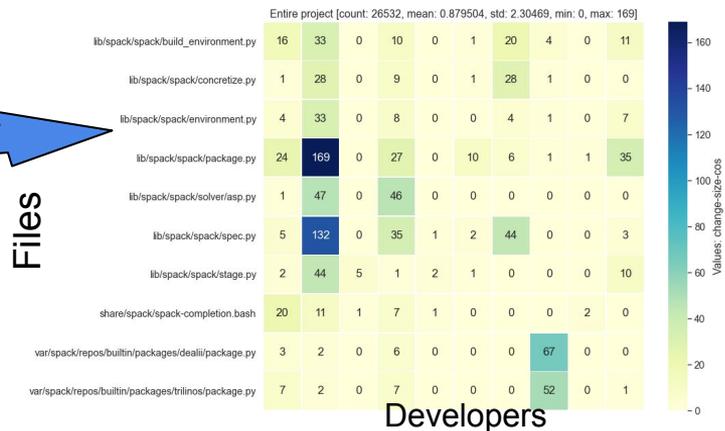


Simple line counts in git commit diffs:

- Patterns such as ‘---+++’ are counted as *edited* lines, e.g., 3 in this example
- Unmatched ‘-’ and ‘+’ lines counted as lines *deleted* and *added*, respectively
- LOCC = edited + deleted + added

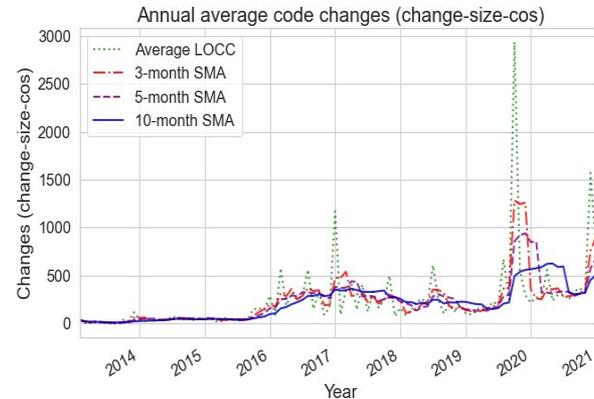
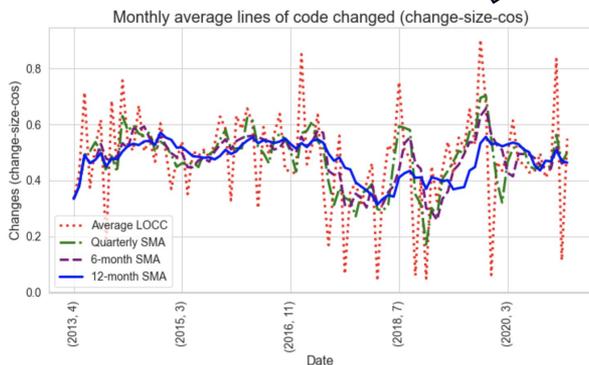
More “intelligent” estimate of the magnitude of changes:

- Collect the *old* and *new* strings corresponding to each commit’s diffs
- Apply text distance metrics (based on the textdistance Python page; ~30 methods)
- E.g., change-size-cos is the cos distance between the vector embeddings of *old* and *new*



Impact of different “change” estimates (cont.)

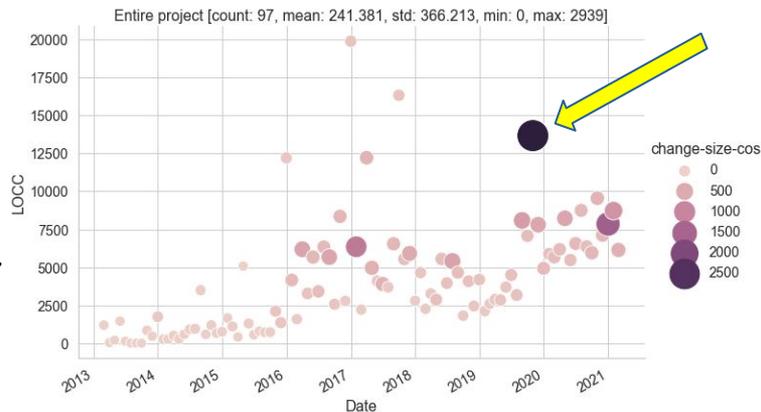
- Time-series git data is messy
 - Moving averages help see trends for any time period



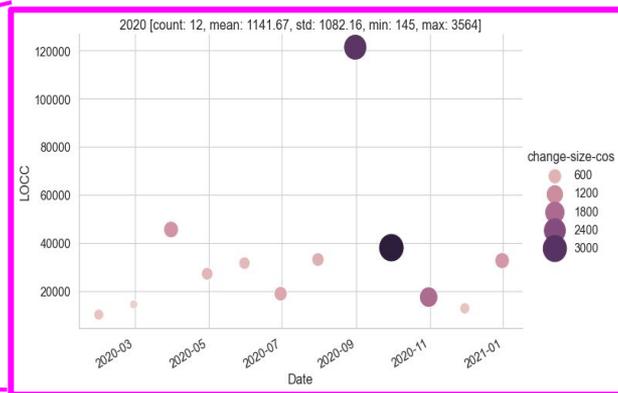
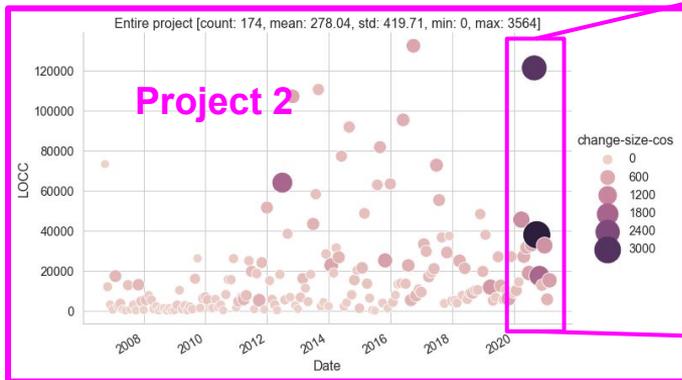
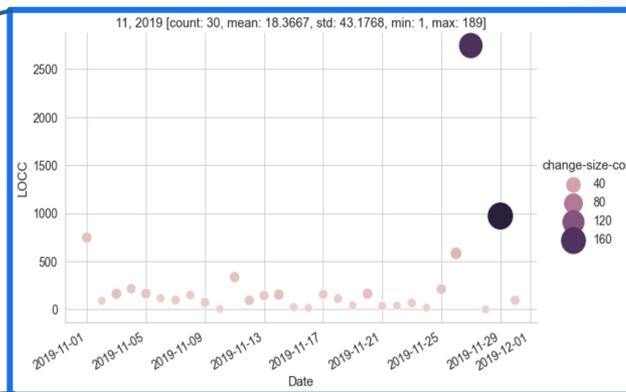
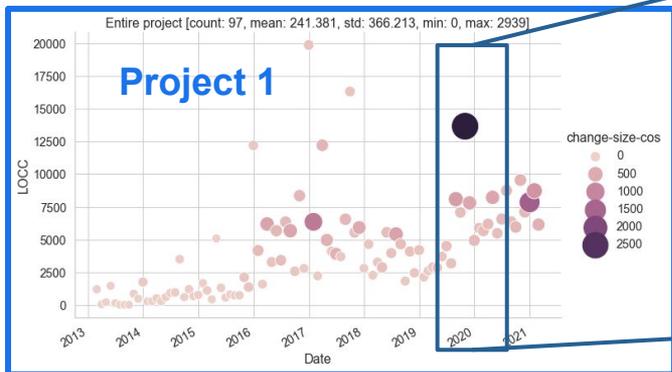
Combining different “change size” metrics



Example on right: simple LOCC totals and cos distance



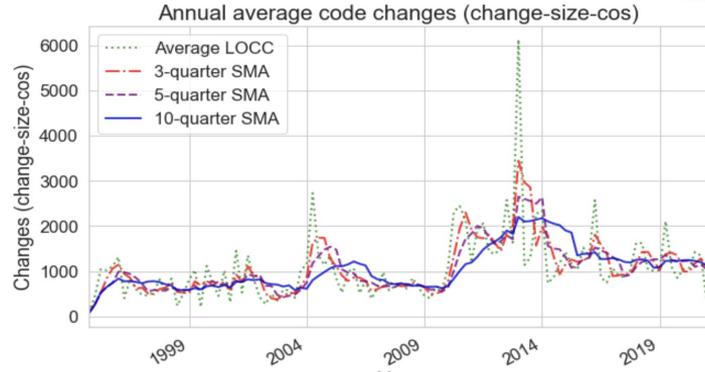
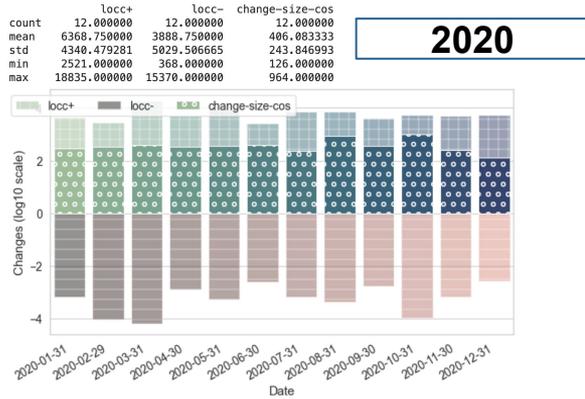
Significant events



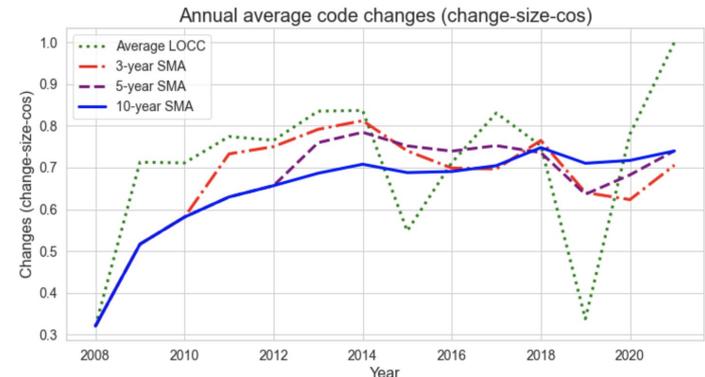
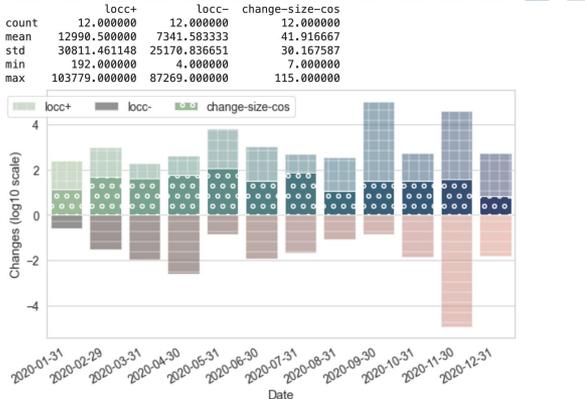
No single view tells the whole story

(project names omitted deliberately)

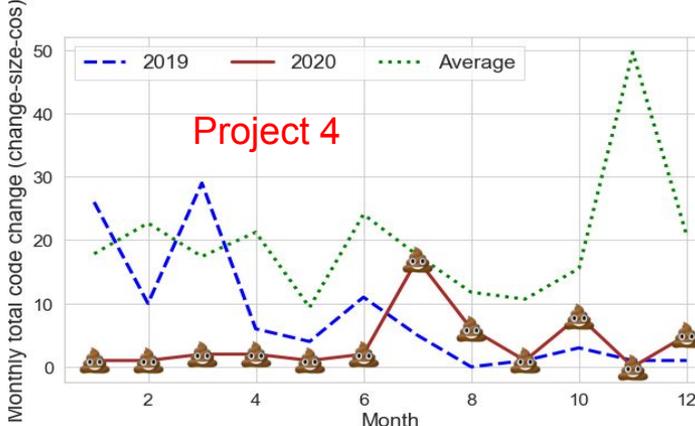
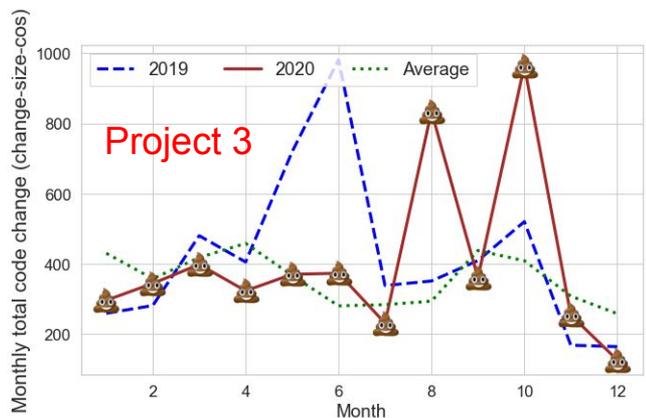
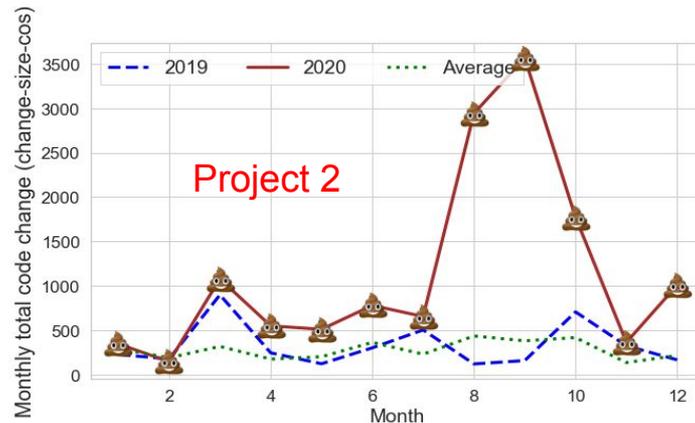
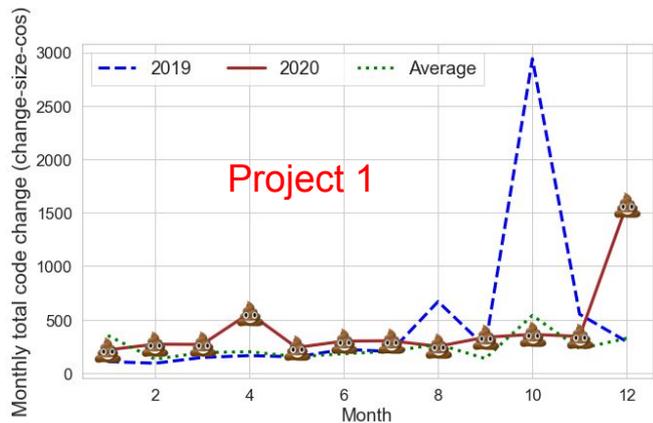
Project 1



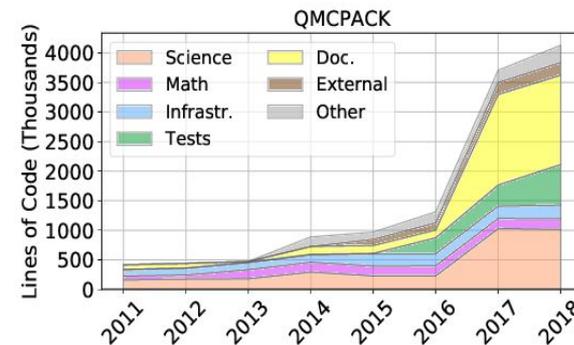
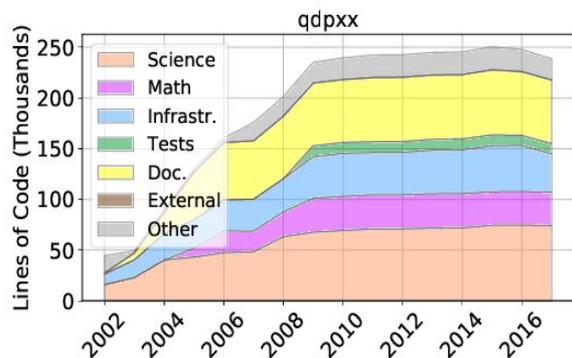
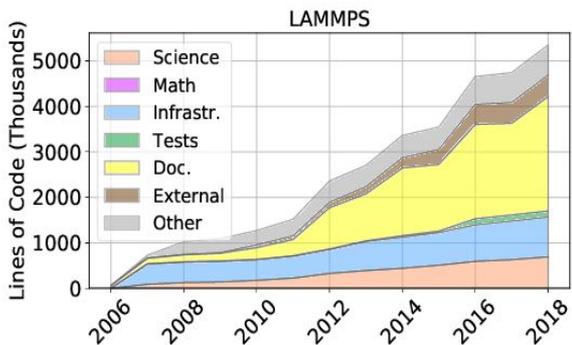
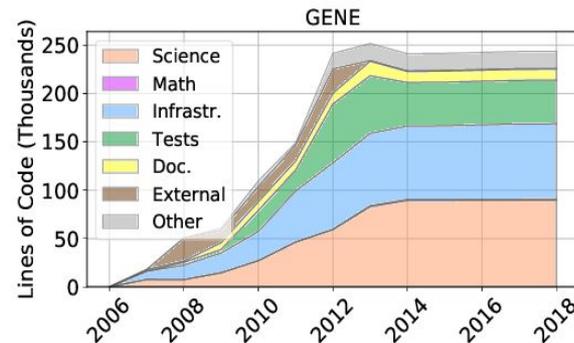
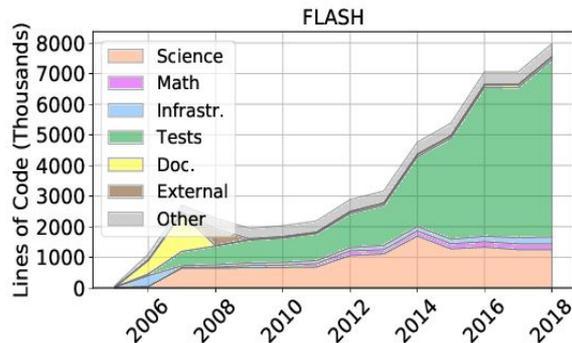
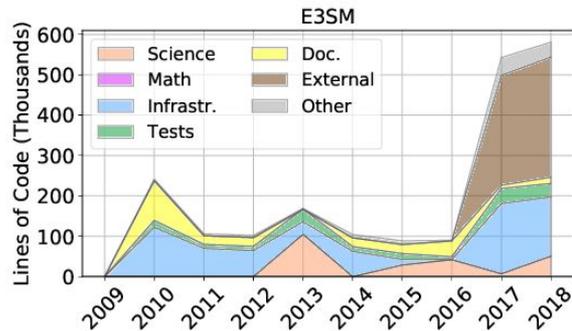
Project 2



How do projects weather interesting times?



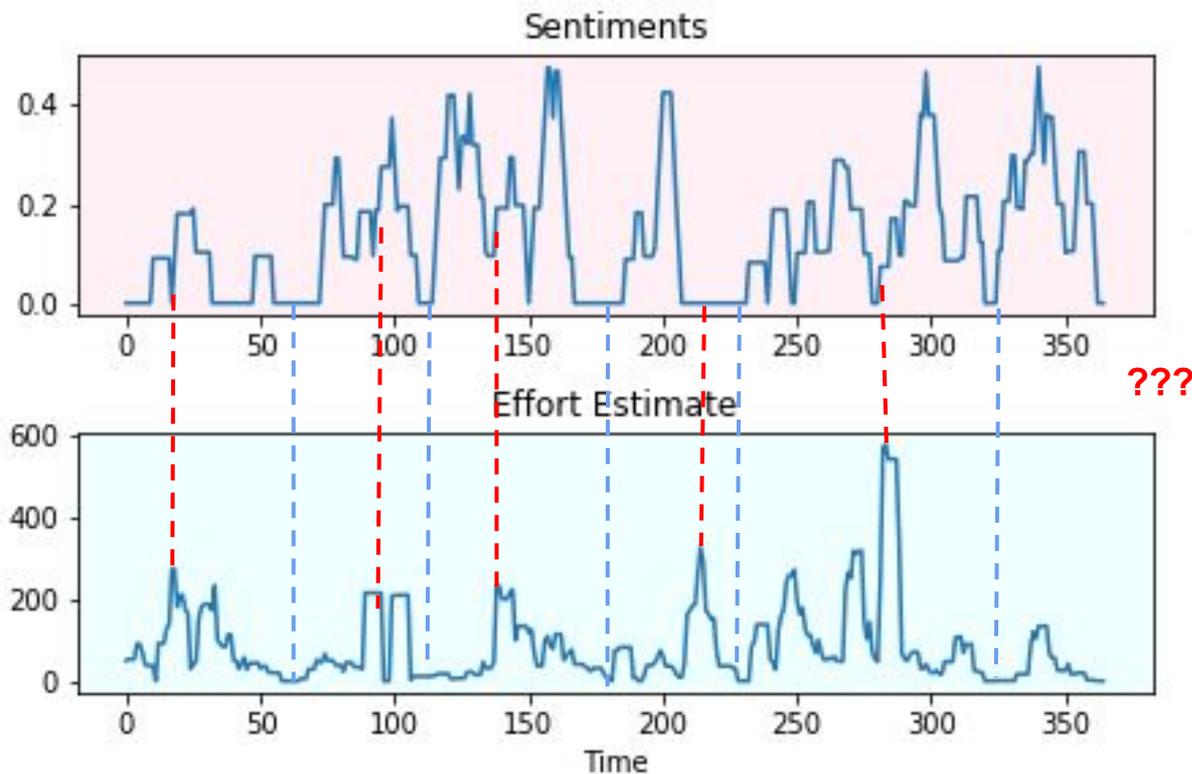
Where is development effort going?



Grannan, A., Sood, K., Norris, B., & Dubey, A. (2020). Understanding the landscape of scientific software used on high-performance computing platforms. *The International Journal of High Performance Computing Applications*. <https://doi.org/10.1177/1094342019899451>

Does developer mood affect productivity?

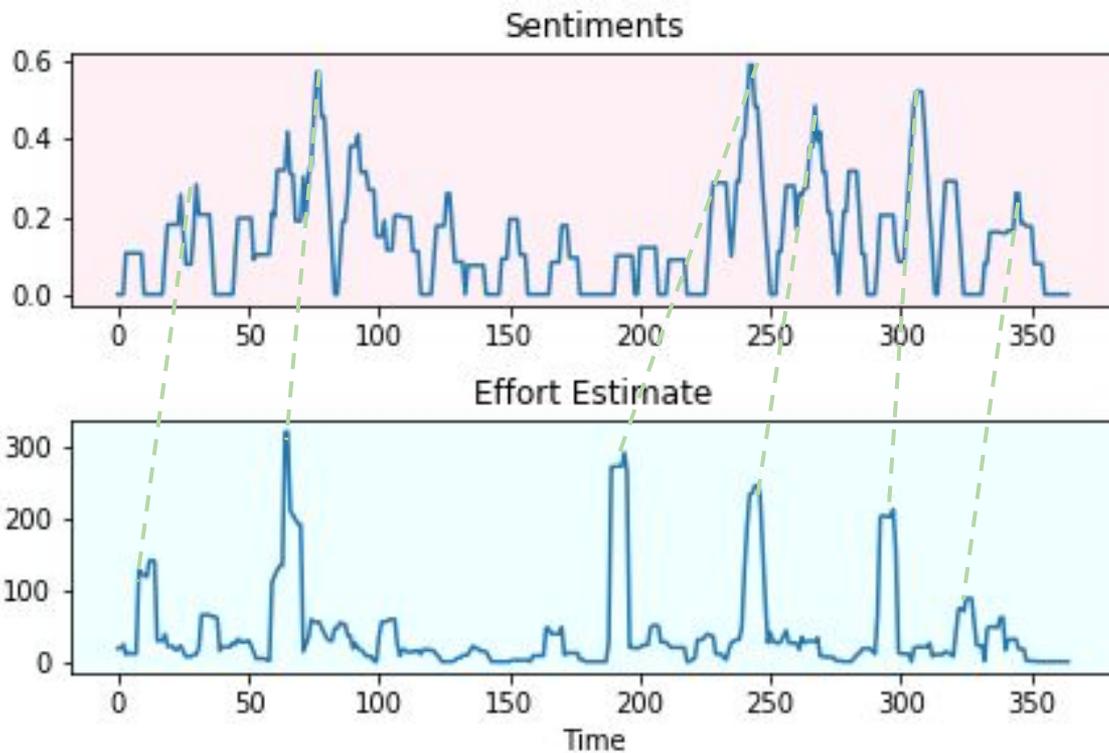
PETSc



...inconclusive, need better analysis

Does developer mood affect productivity?

VTK



Part II: Code Quality

Defects that program analyses can catch

Defects that result from inconsistently following simple, mechanical design rules.

- Security:** Buffer overruns, improperly validated input.
- Memory safety:** Null dereference, uninitialized data.
- Resource leaks:** Memory, OS resources.
- API Protocols:** improper use of APIs, incomplete/incorrect implementations
- Exceptions:** Arithmetic/library/user-defined
- Encapsulation:** Accessing internal data, calling private functions.
- Data races:** Two threads access the same data without synchronization

Key idea: check compliance with (mostly) simple, mechanical design rules.

General-purpose tools for code checking (bugs, style)

➤ C/C++

- Run a bunch of general analyses with **scan-check** (wrapper around `clang --analyze`, which uses the static analyzer below)
 - Minimally invasive, not very customizable
 - Works great with CMake and Autoconf builds
- Clang **static analyzer** component: *extensible* analysis framework for bug finding
 - Can do more complex analyses (path-sensitive, inter-procedural analysis based on a symbolic execution technique)
 - Requires more compiler knowledge to extend
- **Clang-tidy**: *extensible* (libTooling-based) framework for diagnosing typical programming errors or style issues
 - Checking and enforcing of simple coding conventions
 - Modular, provides API for implementing new checks
 - Relatively easy to integrate into Cmake

➤ Fortran

- Flang (compiler front-end to LLVM)
- Fortran-linter (limited)

```
1083 }
1084
1085 #undef FUNC
1086 #define FUNC "Mat_dhTranspose"
1087 void Mat_dhTranspose(Mat_dh A, Mat_dh *Bout)
1088 {
1089     START_FUNC_DH
1090     Mat_dh B;
1091
1092     1 ← 'B' declared without an initial value →
1093
1094     if (np_dh > 1) { SET_V_ERROR("only for sequential"); }
1095
1096     2 ← Assuming 'np_dh' is <= 1 →
1097
1098     3 ← Taking false branch →
1099
1100     Mat_dhCreate(&B); CHECK_V_ERROR;
1101
1102     4 ← Calling 'Mat_dhCreate' →
1103
1104     9 ← Returning from 'Mat_dhCreate' →
1105
1106     10 ← Assuming 'errFlag_dh' is false →
1107
1108     11 ← Taking false branch →
1109
1110     *Bout = B;
1111
1112     12 ← Assigned value is garbage or undefined
1113
1114     B->m = B->n = A->m;
1115     mat_dh_transpose_private(A->m, A->rp, &B->rp, A->cval, &B->cval,
1116                             A->aval, &B->aval); CHECK_V_ERROR;
1117     END_FUNC_DH
1118 }
```

Example development workflow that considers code quality

Example 'make commit' workflow (easy in C/C++, but hopefully possible soon for Fortran):

- clang-format passes and reformats the code
- clang-tidy passes and enforces coding conventions
- clang static analyzer compiles debug and production builds (check errors)
- **xSDK specific analysis** for debug and production build (check errors)
- debug/production builds get compiled and unit tests launched (check errors)
- production build + unit tests run under valgrind (check errors)
- production build with ASan/MSan/TSan gets compiled and unit test launched (check errors)
- production build with --coverage gets compiled and unit test launched against llvm-cov (write unit-test coverage stats)

Our goals and approach

Make it easy(-ish) to define and apply static and dynamic program analysis techniques to identify quality-related problems in xSDK codes.

How? Two parts:

- A. By integrating general **static** and **dynamic** program analyses into the xSDK development process: mainly through documentation and examples.
- B. Creating easy interfaces to custom analyses, examples.

Why?

- Abstraction
 - Elide details of a specific implementation.
 - Capture semantically relevant details; ignore the rest.
- Programs as data
 - Programs are just trees/graphs!
 - ...and we have lots of ways to analyze trees/graphs

Static program analysis is...

Ensure everything is checked the same way.

Examples:

- clang-tidy
- Clang static analyzer

Systematic examination of an abstraction of program state space.

Only track “important” things...

Applies to all possible executions.

Dynamic program analysis is...

Instrumented code only.

Examples:

- Valgrind
- Clang/LLVM sanitizers (better!)

Partial examination of an **abstraction** of a **single** execution path at **runtime**.

Can capture information not available statically.

Applies to specific executions; can miss errors.

Implementation approach

Guiding principle: Whenever possible, leverage existing compiler infrastructure

➤ C/C++:

- Static: Clang Static Analyzer (Clang CFG-based) and clang-tidy interfaces (Matcher-based)
 - Integrate existing checks into library and application builds
 - Implement custom checkers based on coding policies (when available)
 - Produce even higher-level APIs to enable developers to extend checks
 - Check run during package *builds*
 - Questions/challenges:
 - How to deal with non-CMake builds? Spack support?
 - Output formats? (readable, understandable, actionable)
 - Minimize false positives
- Dynamic: Python interfaces to the Clang sanitizer API
 - Used in our current implementation of PETSc development rule checking
 - Requires integration with build system and can be used during *testing*
 - Questions/challenges:
 - How to minimize runtime overheads?
 - Output formats for results? (readable, understandable, actionable)

Implementation approach (continued)

Guiding principle: Whenever possible, leverage existing compiler infrastructure:

➤ Fortran:

- Static: <https://pypi.org/project/fortran-linter/> (extremely limited)
- Flang-based tools evolving (<https://github.com/llvm/llvm-project/tree/main/flang>)
- Dynamic: ?

Yikes?

Example: Using scan-build with hypre¹

```
hypre/src/cmbuild$ scan-build cmake ..  
hypre/src/cmbuild$ scan-build make
```



```
week4 — ssh -AY apollo — 95x20  
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_setup_interp.c.o  
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_setup_rap.c.o  
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_pfmg_solve.c.o  
/home/users/norris/test/hypre/src/sstruct_ls/sys_pfmg_solve.c:157:25: warning: The left operand  
of '>' is a garbage value [core.UndefinedBinaryOperatorResult]  
    if (b_dot_b > 0)  
        ~~~~~ ^  
/home/users/norris/test/hypre/src/sstruct_ls/sys_pfmg_solve.c:168:22: warning: The right operand  
of '/' is a garbage value [core.UndefinedBinaryOperatorResult]  
    if ((r_dot_r/b_dot_b < eps) && (i > 0))  
        ^~~~~~  
2 warnings generated.  
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_semi_interp.c.o  
[ 99%] Building C object CMakeFiles/HYPRE.dir/sstruct_ls/sys_semi_restrict.c.o  
[100%] Linking C static library libHYPRE.a  
[100%] Built target HYPRE  
scan-build: Analysis run complete.  
scan-build: 1136 bugs found.  
scan-build: Run 'scan-view /tmp/scan-build-2021-05-27-073157-25436-1' to examine bug reports.  
norris@apollo:~/test/hypre/src/cmbuild$
```

¹[HYPRE: Scalable Linear Solvers and Multigrid Methods](https://github.com/hypre-space/hypre). <https://github.com/hypre-space/hypre>

Example: hypre (cont.)

```
hypre/src/cmbuild$ scan-view /tmp/scan-build-2021-05-27-073157-25436-1
```

cmbuild - scan-build results

User: norris@apollo
Working Directory: /home/users/norris/test/hypre/src/cmbuild
Command Line: make
Clang Version: clang version 13.0.0 (https://github.com/llvm/llvm-project.git b7911e80d6926f9280ceb23d4e86e25c29370904)
Date: Thu May 27 07:31:57 2021

Bug Summary

Bug Type	Quantity	Display?
All Bugs	1136	<input checked="" type="checkbox"/>
Logic error		
Array subscript is undefined	3	<input checked="" type="checkbox"/>
Assigned value is garbage or undefined	33	<input checked="" type="checkbox"/>
Branch condition evaluates to a garbage value	9	<input checked="" type="checkbox"/>
Dereference of null pointer	378	<input checked="" type="checkbox"/>
Dereference of undefined pointer value	131	<input checked="" type="checkbox"/>
Division by zero	2	<input checked="" type="checkbox"/>
Garbage return value	2	<input checked="" type="checkbox"/>
Result of operation is garbage or undefined	66	<input checked="" type="checkbox"/>
Uninitialized argument value	56	<input checked="" type="checkbox"/>
Unused code		
Dead assignment	387	<input checked="" type="checkbox"/>
Dead increment	10	<input checked="" type="checkbox"/>
Dead initialization	57	<input checked="" type="checkbox"/>
Dead nested assignment	2	<input checked="" type="checkbox"/>

details →

Eek! Too much information, must synthesize a more actionable report.

```
1083 }
1084
1085 #undef FUNC
1086 #define FUNC "Mat_dhTranspose"
1087 void Mat_dhTranspose(Mat_dh A, Mat_dh *Bout)
1088 {
1089     START_FUNC_DH
1090     Mat_dh B;
1091     1 ← 'B' declared without an initial value →
1092     if (np_dh > 1) { SET_V_ERROR("only for sequential"); }
1093     2 ← Assuming 'np_dh' is <= 1 →
1094     3 ← Taking false branch →
1095     Mat_dhCreate(&B); CHECK_V_ERROR;
1096     4 ← Calling 'Mat_dhCreate' →
1097     9 ← Returning from 'Mat_dhCreate' →
1098     10 ← Assuming 'errFlag_dh' is false →
1099     11 ← Taking false branch →
1100     *Bout = B;
1101     12 ← Assigned value is garbage or undefined
1102     B->m = B->n = A->m;
1103     mat_dh_transpose_private(A->m, A->rp, &B->rp, A->cval, &B->cval,
1104                             A->aval, &B->aval); CHECK_V_ERROR;
1105     END_FUNC_DH
1106 }
```

What about project-specific requirements?

Do you need to be a compiler expert to implement new program checks?

Thankfully -- **no!**

Implementation approach: Part B

Develop custom static and dynamic checking based on xSDK requirements.

➤ C/C++

- Static: use and extend existing APIs (Clang static analyzer, clang-tidy); implement custom AST traversals and matchers (more details next)
- Dynamic: use Clang sanitizer APIs; python for simplicity and easy of extensions by xSDK developers

➤ Fortran

- Static: need to write new Flang-based checkers *only* for things that Fortran developers actually care about
- Dynamic: do we need anything?

Coding Rule Violations in HPC Packages: PETSc Case Study

PETSc developer rules: <https://petsc.org/release/developers/style/>

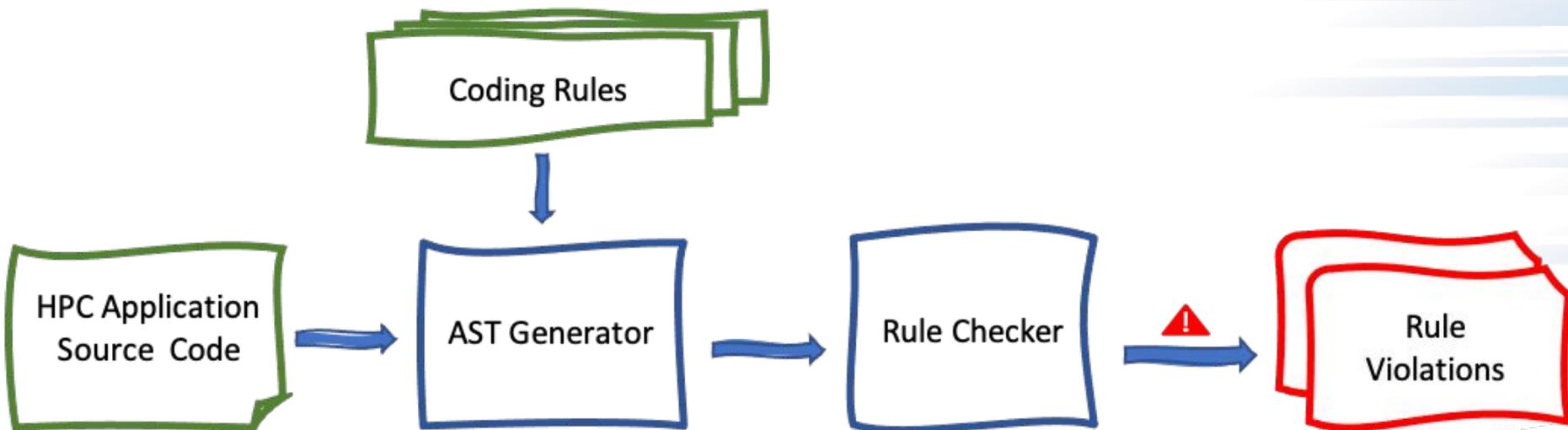
Consider
this
subset:

Rule	Description
Rule-1-A	All function names consist of acronyms or words, each of which is capitalized, for example, <code>KSPSolve()</code> and <code>MatGetOrdering()</code> .
Rule-1-B	All enum types consist of acronyms or words, each of which is capitalized, for example, <code>KSPSolve()</code> and <code>MatGetOrdering()</code> .
Rule-2-A	All macro variables are named with all capital letters. When they consist of several complete words, there is an underscore between each word. For example, <code>MAT_FINAL_ASSEMBLY</code> .
Rule-2-B	All enum elements are named with all capital letters. When they consist of several complete words, there is an underscore between each word. For example, <code>MAT_FINAL_ASSEMBLY</code> .
Rule-3	Functions that are private to PETSc (not callable by the application code) either <ul style="list-style-type: none">• have an appended <code>_Private</code> (for example, <code>StashValues_Private</code>), or• have an appended <code>_Subtype</code> (for example, <code>MatMultSeq_AIJ</code>).
Rule-4	Function names in structures (for example, <code>_matops</code>) are the same as the base application function name without the object prefix and are in small letters. For example, <code>MatMultTranspose()</code> includes the structure name <code>multtranspose</code> .
Rule-5	Names of implementations of class functions should begin with the function name, an underscore, and the name of the implementation, for example, <code>KSPSolve_GMRES()</code> .
Rule-6	Do not use <code>if (rank == 0)</code> or <code>if (v == NULL)</code> or <code>if (flg == PETSC_TRUE)</code> or <code>if (flg == PETSC_FALSE)</code> . Instead, use <code>if (!rank)</code> or <code>if (!v)</code> or <code>if (flg)</code> or <code>if (!flg)</code> .
Rule-7	Do not use <code>#ifdef</code> or <code>#ifndef</code> . Rather, use <code>#if defined(...)</code> or <code>#if defined(...)</code> . Better, use <code>PetscDefined()</code> .

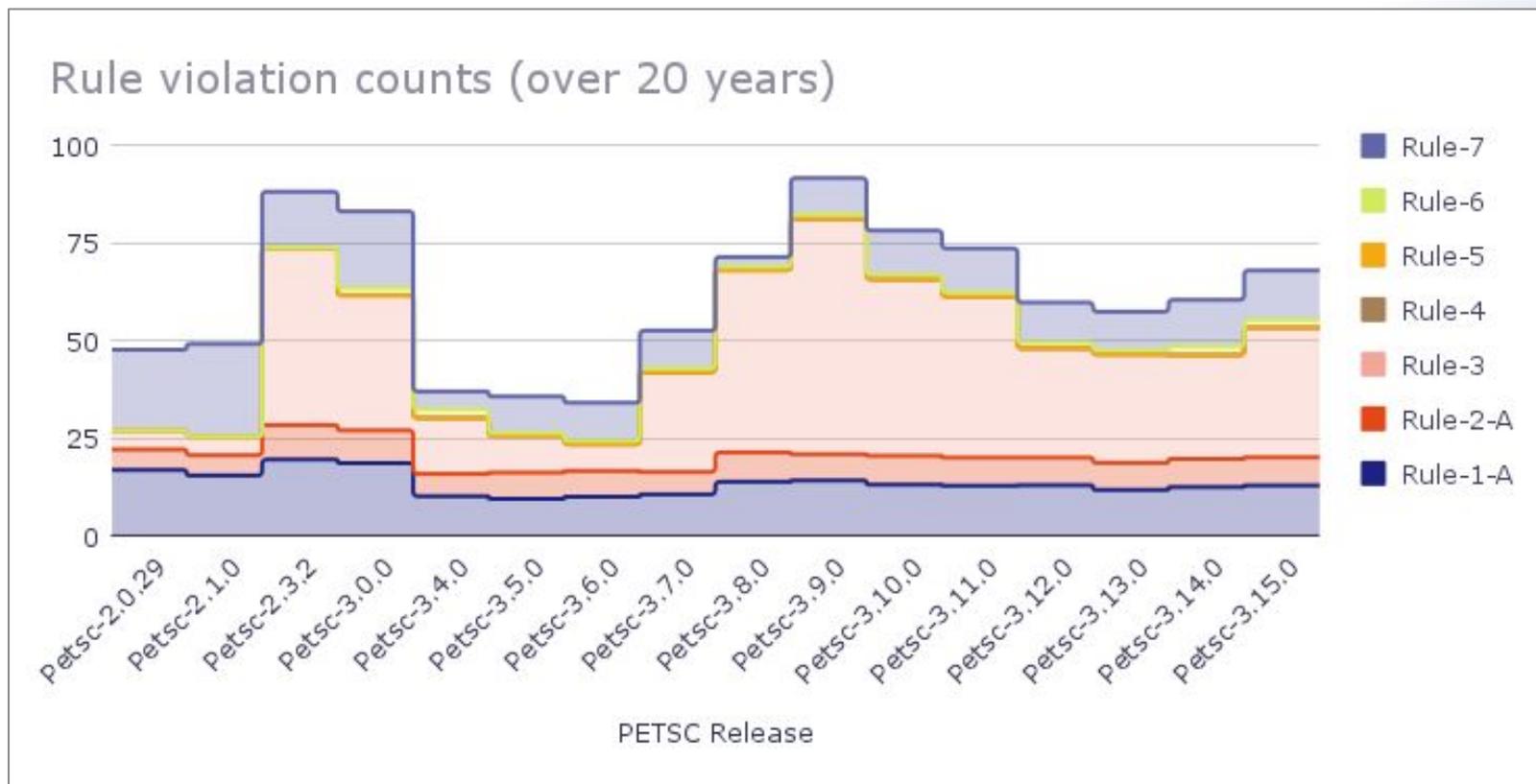
Examples of developer rule violations (PETSc 3.15)

PETSc Rule	PETSc Construct	Description	path	Line	Column
Rule-1	Function definition in the library	PetscErrorCode PETSCMAP1(VecScatterBeginMPI3Node)(VecScatter ctx,Vec xin,Vec yin,InsertMode addv,ScatterMode mode)	~/petsc-3.14.3/src/vec/vscat/impls/mpi3/vp scat.h	249	16
Rule-2	Macro in the library	#define mpi_reduce_scatter PETSC_MPI_REDUCE_SCATTER	~/petsc-3.14.3/include/petsc/mpiuni/mpiuni fdef.h	118	2
Rule-3	Function declaration in the library	PETSC_INTERN PetscErrorCode MatFactorFactorizeSchurComplement_Private(Mat);	~/petsc-3.14.3/include/petsc/private/matim pl.h	494	29
	Function declaration in the library	PETSC_EXTERN PetscErrorCode MatFactorFactorizeSchurComplement(Mat);	~/petsc-3.14.3/include/petscmat.h	1245	29
Rule-4	Function definition in the library	PETSC_EXTERN PetscErrorCode DMDAVecGetArray(DM,Vec,void *)	~/petsc-3.14.3/include/petscdmda.h	113	29
	Function call in the application	ierr = VecGetArray(y,yv)	~/petsc-3.14.3/include/petscvec.h	545	10
Rule-5	Function call in the library	ierr = PetscFEPushforwardGradient(fe, fegeom, 1, interpolantGrad);	~/petsc-3.14.3/include/petsc/private/petscf eimpl.h	332	10
Rule-6	If in the library	if (p == 0) return node;	~/petsc-3.14.3/src/dm/impls/plex/gmshlex.h	231	3
Rule-7	Macro in the library	#ifndef PETSC4PY_COMPAT_MUMPS_H	~/petsc-3.14.3/src/binding/petsc4py/src/incl ude/compat/mumps.h	1	1

Checking for violations of the PETSc developer rules



Example results for a subset of the PETSc rules



Capabilities summary

Type of Data/Analysis	Database	Examples	Repository Location
Git data: commits, changes (lines, files, etc.)	✓	✓	github.com/HPCL/ideas-uo/
Github and Gilab issues and associated metadata	✓	✓	github.com/HPCL/ideas-uo/
Code quality checkers (dynamic & static)	✗	✓	<ul style="list-style-type: none">• github.com/HPCL/code-analysis (dynamic)• github.com/HPCL/llvm-project/tree/xsdk-uo/clang-tools-extra/clang-tidy/petsc (static)
Mailing lists	✓	✓	Not publicly available yet, contact norris@cs.uoregon.edu

Summary

- We introduced a **flexible**, **efficient**, and **usable** software framework for **acquiring**, **storing**, **manipulating**, and **visualizing** development-related data.
- We demonstrated a few of its capabilities here; an ever growing number of patterns and other analyses are continuously being developed.
 - **Contributions and/or requests welcome!** <https://github.com/HPCL/ideas-uo/>
- Acknowledgments: DOE ECP IDEAS Productivity Project
 - Carter Perkins, Bosco Ndemeye, Stephen Fickas, University of Oregon
 - Armando Acosta and Kanika Sood, California State University, Fullerton
 - Anshu Dubey and Lois Curfman McInnes, Argonne National Laboratory

Thank you!

ECP projects that may be present in examples in this webinar: Spack, LAMMPS, PETSc, Nek5000 E3SM, QMCPACK, QDPXX, LATTE NAMD, HYPRE, fast-export, Enzo, TAU2, xpress-apex, LATTE, NWChem