CODEMINE:

Building a Software Development Data Analytics Platform at Microsoft

Jacek Czerwonka, Microsoft

Nachiappan Nagappan, Microsoft Research

Wolfram Schulte, Microsoft

Brendan Murphy, Microsoft Research

// The process Microsoft has gone through developing CODEMINE—a software development data analytics platform for collecting and analyzing engineering process data—includes constraints, and pivotal organizational and technical choices. //

EARLY, TRUSTWORTHY DATA available at the required frequencies lets engineers and managers make data-driven decisions that can enable the success of the software development process to deliver a high-quality, on-time software system. At Microsoft, several teams use data to improve processes. Examples include

- trend monitoring and reports on development health,[1]
- risk evaluation and change impact analysis tools,[2]
- version control branch structure optimization,[3]
- social-technical data analysis,[4] and
- custom search for bugs and debug logs, speeding up investigations of new issues.[5]

When reviewing these and other solutions from our existing portfolio of tools, our teams realized that even though each solution is unique in its intended purpose and the way it improves the engineering process, there are commonalities of inputs, outputs, and methods used among the tools. For example, a majority of the reviewed tools need similar input data: source code repositories and system binaries, defect databases, and organization hierarchies.

In late 2009, a team at Microsoft was established to explore and implement a common platform, CODEMINE, for collecting and analyzing engineering process data from across a diverse set of Microsoft's product teams. CODEMINE quickly became pervasive and is now deployed in all major product groups at Microsoft. This project wasn't done for the sake of research or academic impact but was actually deployed in Microsoft and has hundreds of users. Currently, CODEMINE is deployed in all major Microsoft product groups: Windows, Windows Phone, Office, Exchange, Lync, SQL, Azure, Bing, and Xbox.

This article presents the motivation, challenges, solutions, and most important, the lessons learned by the CODEMINE team to aid in replicating such a platform in other organizations. We hope our design rationale can help others who are building similar analytics platforms.

Data Sources and Schema

Figure 1 depicts a high-level schema of the repositories and types of artifacts mined by CODEMINE. In terms of both volume and frequency of change, source code repositories are the largest sources of engineering data for a company like Microsoft. They contain information on a variety of source code–related artifacts divided into data describing its state, composition, and high-level attributes, as well as data describing ongoing code changes. In the

0740-7459/13/$31.00 © 2013 IEEE

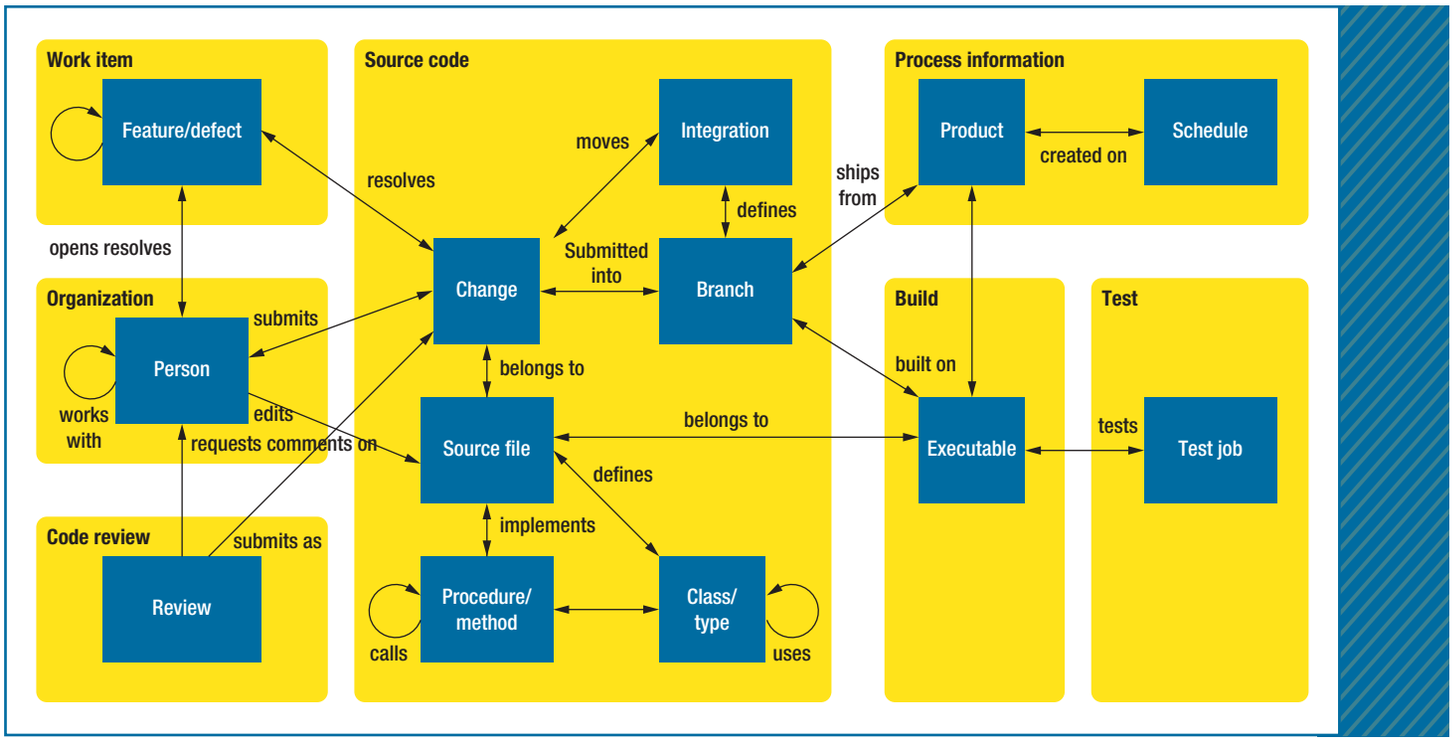

FIGURE 1. The types of data CODEMINE platform collects. Artifacts are cross-referenced as much as possible, allowing queries against CODEMINE to go beyond an individual repository.

former category, the primary concepts are source files and their attributes: total size, size of code versus comments, implemented methods, and defined classes or types. In the latter, concepts of a change, a branch, and an integration characterize the team's output over time.

Another large and important body of data resides in work item repositories. These typically encompass both features and defects, both types of which are often tightly linked to source code changes. It's a bidirectional relationship—features and defects are both a trigger for as well as a cause of source code changes.

Data on builds describes the composition of the final software product and also allows us to map source code to the resulting executable. Code reviews and tests complete the picture of the engineering activity, taking into account the two most common software verification and validation activities. Organization information and process information (such as release schedules and development milestones) are also a part of CODEMINE. They provide context for the engineering activity, the code being developed, and all activities around that.

As Figure 1 depicts, artifacts are cross-referenced as much as possible, allowing queries against CODEMINE to go beyond an individual repository.

Architecture

Figure 2 describes the CODEMINE platform's high-level architecture. More than one instance currently exists; all conform to the same blueprint. We're assuming a high degree of commonality in the data stored in and accessible from each instance of the CODEMINE data platform; however, each instance might have slightly different capabilities, in terms of both data stored and analytics that execute on it. Yet, client applications will be able to run on the data platform as long as the data they need is present, ideally scaling their capabilities on the basis of which data is actually present. If an application can't run on a particular instance of the data platform, it will be able to fail gracefully.

Data Store

The core element of the data platform is the data store. It's a logical concept realized as a collection of data sources—typically databases but also file shares with either text or binary files. These data sources don't have to be colocated but are likely to remain geographically

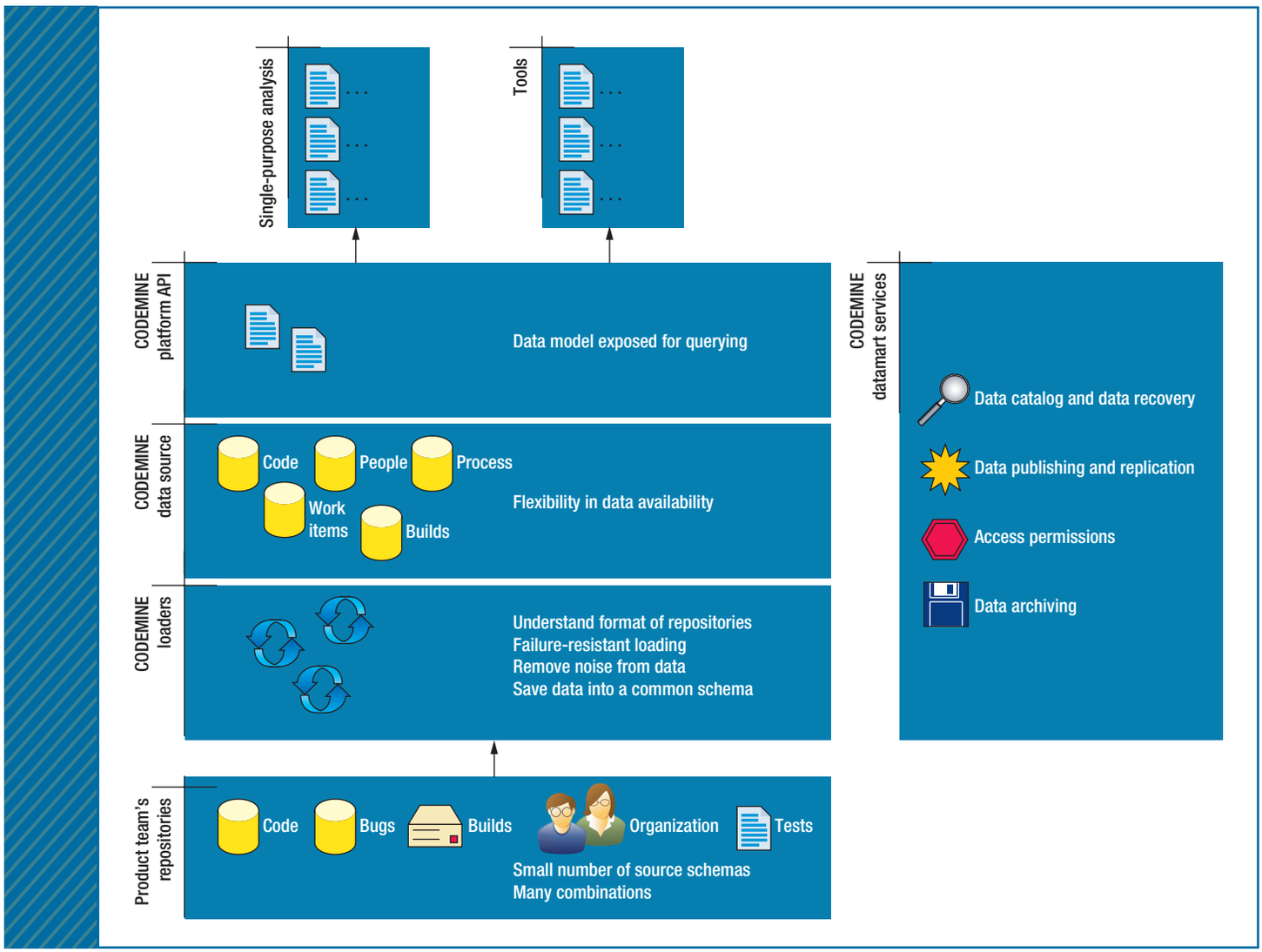

FIGURE 2. High-level architecture of a CODEMINE data platform instance.

close to the raw data they cache consistent with individual product group data and security policy. It's not necessary for all data platform deployments to have the same data sources. Applications use the data catalog service to query for presence and logical location (such as a connection string or file share name) of specific pieces of data.

Data Loaders

Data loaders are modules of code that read raw data and directly put it into the data store. They understand the schema of the raw data source they're querying from. Data loaders are built to be as independent and decoupled from one another as possible.

The data collection workflow takes care of orchestrating data collections, enforcing any dependencies, and ensuring collections happen in the correct order. The workflow will be defined in close cooperation with product groups and adheres to the "pull once" model of data collection as closely as possible.

Platform APIs (Data Model)

CODEMINE has a standard set of interfaces that expose data from the data platform. The interfaces target most common entities such as code, defects, features, tests, people, and their attributes and relationships. The most common usage patterns should be realized through this data model.

Applications that make use of the data platform will most often follow this pattern:

1. Query the data catalog to ensure that the needed data exists. Fetch connection strings to data sources or URLs to needed services.
2. Tailor functionality depending on the available data.
3. Connect to services and query for data through the data model.
4. Display data.

The data model is the preferred way to access the data stored in the data platform. It needs to be expressive enough to support the data needs of the productized solutions. However, for purposes of specialized queries, one-off research tasks, or prototyping, access to interfaces exposed by individual data sources is also available.

Platform Services

Platform services encompass a variety of features related to data cataloging, security and access permissions, event logging, data archiving, and data publishing.

Each part of the data platform system needs to be able to log events to a common place. Reasons for logging include health monitoring and trending, data access auditing, execution tracing, and alerting in failure cases.

Product groups need the ability to control access to their cached data the same way they control access to raw data sources. The security policy module must be able to understand the security configuration systems used by product groups, query the security policies at the right frequency, and apply them to both stored data and interfaces accessible from outside the data platform. Currently, data platform instances are protected by individual and separate security groups.

Data Platform Usage Scenarios

In the process of creating the platform and opening it up to both the Microsoft internal research community and product groups, three distinct patterns of data use emerged:

- *As a data source for a reporting tool or methodology that's part of a product team's process.* When a product team uses the CODEMINE platform and the client application in production, this usage pattern requires data freshness and reliability of data acquisition and analysis as well as operational uptime and efficiency to get to data.
- *For one-time, custom analysis focusing on answering a specific question.* Although the data might not be stored in a way that's optimized for a particular query, the fact that the data is available at all and easy to access (compared to accessing raw data sources for the same data) makes CODEMINE the go-to data source when a product team needs to make a decision related to their product, process, or organization.
- *To enable new research.* Data from each product team, and especially from across product teams, is a compelling source of information and inspiration for new lines of research.

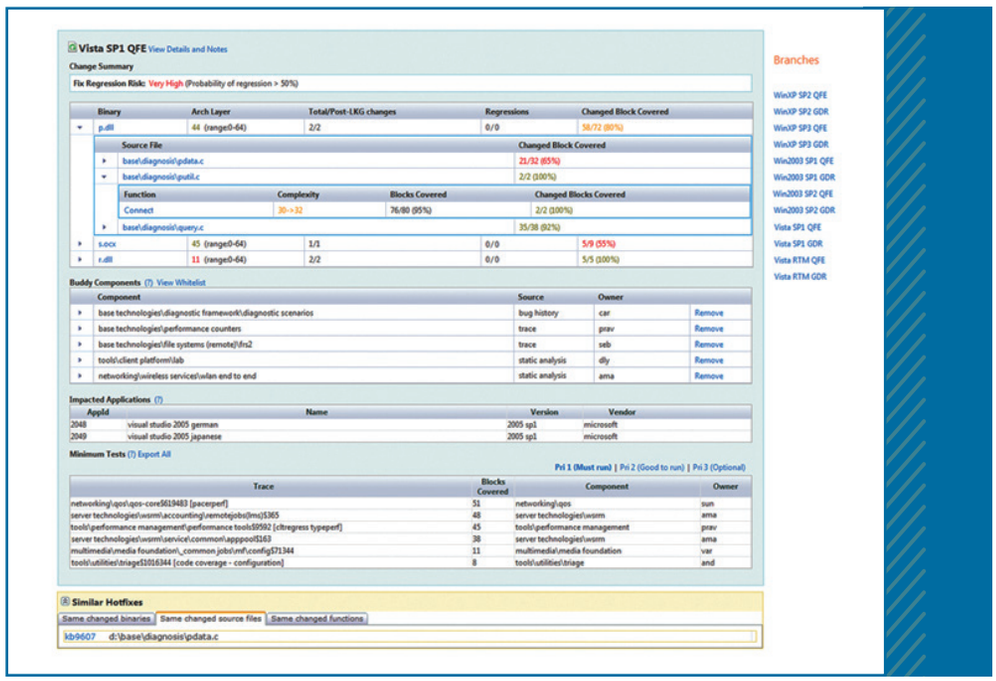

FIGURE 3. CRANE tool screenshot.

What follows are examples from each of these categories.

Example 1: Mature Research Encoded into a Tool

Change is a fundamental unit of work for software development teams that exists regardless of whether a product is a traditional boxed version or a service or whether a team uses an agile process or a more traditional approach.

Making postrelease changes requires a thorough understanding of not only the architecture of the software component to be changed but also its dependencies and interactions with

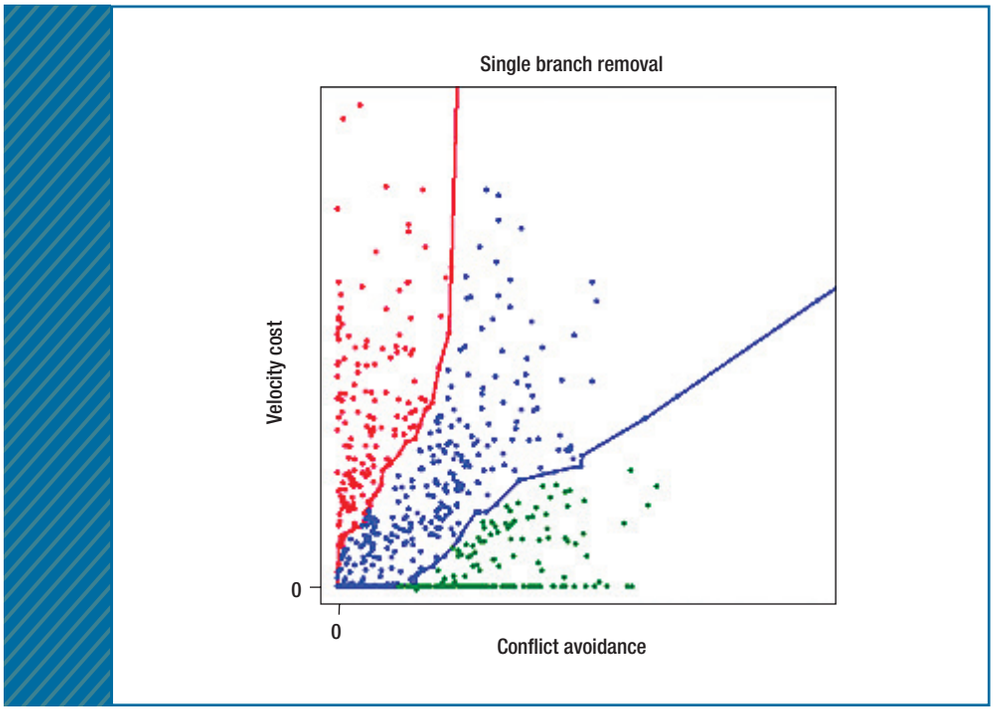

FIGURE 4. Velocity versus conflict avoidance. Red dots indicate branches that aren't useful, green dots indicate branches that are useful, and blue dots indicate branches with mixed utility.

other system components. Testing such changes in reasonable time and at a reasonable cost is a problem because an infinite number of test cases can be executed for any modification.[2] Furthermore, they're applicable to hundreds of millions of users; even the smallest mistakes can translate to very costly failures and rework.

CRANE is a failure prediction, change risk analysis, and test prioritization system at Microsoft Corporation that leverages existing research for the development and maintenance of the Windows operating system family.[2]

CRANE is built on top of the CODEMINE infrastructure, as shown in Figure 2 on the top layer, where the tools exist leveraging the CODEMINE platform. The CODEMINE data platform constantly monitors changes happening in the source code repository and can cross-reference these with features, defects, people, code reviews, and auxiliary data such as code coverage. CRANE is able to use this data, and consequently, teams can automatically receive information on change composition, associated bugs, similar changes, involved people, and possible risks and recommend risk-mitigation steps.

CRANE is able to not only surface information about changes but also provide interpretation via overlaying coverage data and statistical risk models to identify the most risky and least covered parts of a change. Figure 3 shows a snapshot of a CRANE analysis, which identifies change, coverage, dependency, people, and prior bug information. It allows engineers and engineering managers to focus their attention on the most failure-prone parts of their work. Through use of code coverage data and a maximum-coverage/minimum-cost algorithm, CRANE is able to recommend specific, high-value tests.

The system has already been successfully deployed in Windows, and pilots are underway in other product teams.

Example 2: Ad Hoc Analysis for Decision Making

Here's a simple but very important question: Is code coverage effective, and is there a code coverage percentage at which we should stop testing?

We analyzed code coverage of multiple released versions of Microsoft products and correlated branch and statement coverage with postrelease failures. There was a strong positive correlation between coverage and failures. From discussions with the relavant teams, we found out that there are several reasons for the existence of this inverse relationship:

- code covered doesn't guarantee that the code is correct;
- having 100 percent code coverage doesn't mean the system will have no failures—rather, it means that bugs could be found outside anticipated coverage scenarios; and
- each time a fix is done, a test case is written to cover the fix (often, changed binaries might therefore have high code coverage simply because they have been modified several times).

This finding led us to a follow-up study on the use of code coverage in conjunction with code complexity (for example, cyclomatic complexity and class coupling) as a better indicator of code quality. In addition, we were able to benchmark our results with external organizations such as Avaya to compare and contrast our results.[6] Studies of unit testing show its increased effectiveness in obtaining high-quality code

because it eliminates the need for testers to find the category of bugs that could be more easily found by developers and lets them focus more on scenario testing.[7]

Example 3: Use of Data in New Research

Many companies use branches in version control systems to coordinate the work of hundreds to thousands of developers building a software system or service. Branches isolate concurrent work, avoiding instability during development. The downside is an increase in time for changes to move though the system. So, can we determine the optimal branch structure, guaranteeing fast code velocity and a high degree of isolation? Answering this question isn't only important to Microsoft but also to other commercial companies and the research community.

Toward this end, we performed various experiments on simulating different branch structures.[3] For example, we replayed the check-in history of several product groups, assuming specific branches didn't exist. Under these conditions, all changes hierarchically roll up to a higher-level branch, and we can detect conflicts by identifying files getting modified together. The resulting graph (see Figure 4) plots the cost of a branch versus its value as a factor, isolating parallel lines of development. In Figure 4, red dots indicate branches that aren't useful—that is, adding velocity and not providing much conflict avoidance. Green dots indicate branches that are useful, and blue dots indicate branches with mixed utility. Branch structures are created in context and to suit needs of a specific product and organization; such branch evaluation lets teams identify the cost paid for the benefit and identify parts of the branch tree that should be restructured.[3]

We also analyzed the architectural structure of Windows (for both Vista and Windows 7) and observed that a branch structure that aligns with the team's organizational structure leads to fewer postrelease failures than branches aligned to the product's architectural layering.[8]

Lessons on Replicating CODEMINE

One of our primary goals in this article is to help other organizations replicate the work we're doing with CODEMINE to build their own data analytics platform. We've compiled a list of suggestions from our experience that would assist in replicating our CODEMINE effort and some things for other organizations to think of differently when building their platform.

> Branch evaluation lets teams identify the cost paid for the benefit

Create an Independent Instance for Each Product Team in the Data Platform

Easy partitioning, the ability to constrain access, and the ability to move parts of the infrastructure greatly assisted us in creating independent instances.

Have Uniform Interfaces for Data Analysis

Even though multiple instances will exist, applications need to rely on a common set of services, APIs, or a stable schema present in each. The data platform interfaces' evolution must be done very carefully; preserving backward compatibility should be of primary concern. This also greatly helps when you build an application once and can redeploy it multiple times across several data instances.

Encode Process Information

Process information, including release schedule (milestones and dates), organization of code bases, team structure, and so on, is very important to provide better context—for example, why is there a sudden spike in bugs (more users added), sudden spike in code churn (code integration milestone), and so on? At Microsoft, this information isn't present in one place or tool. It might pop up in project tracking, a code repository, and bug-tracking tools or be configured with some level of customization to interpret this. Organizations should make plans to embed this information in the system to provide valuable metadata.

Provide Flexibility and Extensibility for Collected Data and Deployed Analytics

Product teams have varying requirements and need the ability to define which data and metadata are stored in the data platform and how they're analyzed; this will allow teams to best reflect their existing processes or enable new ones. For example, one team might decide to add customer user data to their instance of their data store. The system should be able to fully support such extensions.

Allow Dynamic Discovery of Data Platform's Capabilities by Application

Each application relying on the data platform needs the ability to identify capabilities of a particular data platform instance and adjust its function accordingly. For example, some product groups collect and archive historical

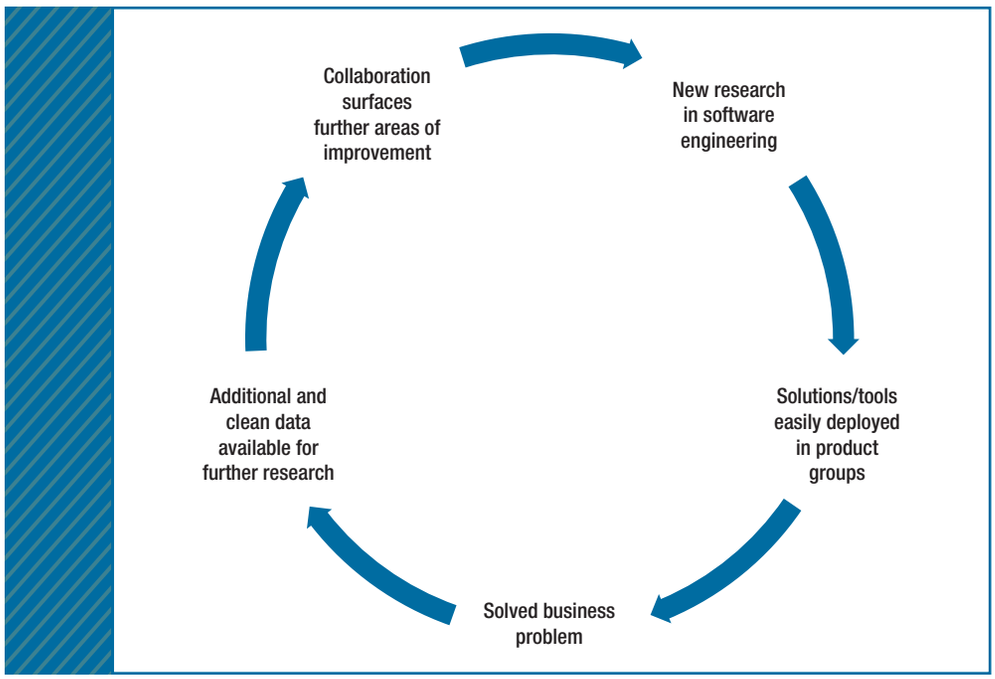

FIGURE 5. Cycle of collaboration and data availability.

code coverage data and some choose not to. The tools must be able to seamlessly scale their functionality down if code coverage data isn't available for a particular product group.

Support Policies for Security, Privacy, and Audit

The data platform must allow for setting authorization, authentication, privacy, and audit policies to reflect security requirements and policies of the product group or the data owner. As a general rule, information leaving the data platform will be accessible only to people who were granted permission to the original raw data coming in; however, stricter rules might apply to subsets of data.

Allow Ongoing Support and Maintenance Outside of CODEMINE

In most cases, product teams eventually take over ownership and operations for their respective data platform instances. To ensure a smooth transition, the data platform must adhere to rules of a well-behaved service defined by operations teams. Resiliency to failure, retry logic, logging of fatal and nonfatal errors, health monitoring, and notifications should be built in.

Host as a Cloud Service

Based on need, economic considerations, load, and availability requirements, carefully evaluate the necessity to host the service on the cloud or on traditional servers. Overengineering always leads to wasted effort.

Know the Data Platform Might Not Fulfill All Data Needs

The data platform will be scoped to provide data that's used by several client applications—that is, there must be a level of commonality of inputs for the platform to start serving the data. However, applications can still access other, more specialized data sources and federate with the data platform as their needs dictate.

Innovate at the Right Level of the Stack

Use mature foundational technology and existing programming skills. As much as possible, we try to use the operating system, storage, and database platform technology that's mature and already part of Microsoft's stack to avoid spending time innovating, for example, at the level of raw storage or methods of distributed computation. Instead, we focus on data availability, accurate data acquisition, data cleaning, abstracting representation of the engineering process, and data analysis. In terms of accessing data, we need to ensure any new programming models used are absolutely necessary for the task so we don't create artificial barriers of entry for users of our data.

We've observed that once data is easily accessible, new usage scenarios open up; for instance, CODEMINE is currently being used to understand onboarding processes, optimize individual processes (like build), and optimize overall code flow.

Another significant goal of the CODEMINE platform is enabling future research and analysis. Figure 5 explains the cycle of data availability where new research in software engineering spawns new solutions to be deployed in product groups. These solutions can be used to solve large business problems and enable additional research with the scaling out of the engineering work. This further strengthens the collaboration and opens new avenues for collaboration and again leads to new research ideas.

As a way to propagate the ideas of data-driven decision making, we recently started a virtual community focused on sharing questions, solutions, methods, and tools related to engineering process data analysis. It is a cross-disciplinary group of product team

members and researchers with experience and backgrounds in empirical software engineering, data analysis, and data visualization. The group's goal is to emphasize data-driven decision making in our teams and to equip product teams with relevant guidelines, methods, and tools. As we realize our goals, the CODEMINE data platform often serves as the common denominator in our community activities.

References

1. N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM, 2005, pp. 284–292.
2. J. Czerwonka et al., "CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice—Experiences from Windows," *Proc. 4th Int'l Conf. Software Testing, Verification and Validation* (ICST 11), IEEE CS, 2011, pp. 357–366.
3. C. Bird and T. Zimmermann, "Assessing the Value of Branches with What-If Analysis," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), ACM, 2012, pp. 45–54.
4. C. Bird et al., "Putting It All Together: Using Socio-technical Networks to Predict Failures," *Proc. 20th Int'l Symp. Software Reliability Eng.* (ISSRE 09), IEEE CS, 2009, pp. 109–119.
5. B. Ashok et al., "DebugAdvisor: A Recommender System for Debugging," *Proc. 7th Joint Meeting European Software Eng. Conf. and ACM SIGSSOFT Symp. Foundations of Software Eng.* (ESEC/FSE 09), ACM, 2009, pp. 373–382.
6. A. Mockus, N. Nagappan, and T.T. Dinh-Trong, "Test Coverage and Post-verification Defects: A Multiple Case Study," *Proc. 3rd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 09), IEEE CS, 2009, pp. 291–301.
7. L. Williams, G. Kudrjavets, and N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft," *Proc. 20th Int'l Symp. Software Reliability Eng.* (ISSRE 09), IEEE CS, 2009, pp. 81–89.
8. E. Shihab, C. Bird, and T. Zimmermann, "The Effect of Branching Strategies on Software Quality," *Proc. Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 12), ACM, 2012, pp. 301–310.

ABOUT THE AUTHORS

JACEK CZERWONKA is a principal software architect in the Tools for Software Engineers group at Microsoft. His research interests include software testing and quality assurance, systems-level testing, pairwise and model-based testing, and data-driven decision making on software projects. Czerwonka received his MSc in Computer Science from Technical University of Szczecin. Contact him at jacekcz@microsoft.com.

NACHIAPPAN NAGAPPAN is a principal researcher in the Empirical Software Engineering group at Microsoft Research. His research interests include software analytics, focusing software reliability, and empirical software engineering processes. Nagappan received a PhD in computer science from North Carolina State University. Contact him at nachin@microsoft.com.

WOLFRAM SCHULTE is an engineering general manager and principal researcher at Microsoft. His research interests include software engineering, focusing on build, modeling, verification, test, and programming languages, ranging from language design to runtimes. Schulte received a PhD in computer science from the Technical University of Berlin. Contact him at schulte@microsoft.com.

BRENDAN MURPHY is a principal researcher at Microsoft Research. His research interests include system dependability, encompassing measurement, reliability, and availability. Contact him at bmurphy@microsoft.com.