Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study

Carlene Lebeuf
University of Victoria, Canada
clebeuf@uvic.ca

Elena Voyloshnikova
Microsoft, USA
elvoyl@microsoft.com

Kim Herzig
Microsoft, USA
kimh@microsoft.com

Margaret-Anne Storey
University of Victoria, Canada
mstorey@uvic.ca

***Abstract*—Today's build systems distribute build tasks across thousands of machines, reusing cached build results whenever possible. But despite the sophisticated nature of modern build tools, the core software architecture of the system under build defines the lower bound for how fast the system can compile. Long, consecutive build chains or slow individual build targets can introduce expensive compilation bottlenecks. Further, the growing complexity of both build systems and software systems under build makes comprehending, debugging, and optimizing build performance a significant challenge faced by many software engineers. We present a design study to describe and help mitigate the cognitive challenges faced by software engineers that use modern, cached, and distributed build systems. We characterize the performance analysis process and identify the main stakeholders involved, key usage scenarios, and elicit important requirements for tool support. We propose an interactive *BuildExplorer* tool for understanding, optimizing, and debugging cached and distributed build sessions, justifying our design decisions among alternative solutions. Our novel solution is evaluated through usage scenario walkthroughs, iterative deployments of the tool in the field, and a user study.**

***Index Terms*—Build Systems, Build Optimization, Build Debugging, Software Visualization, Design Science**

I. Introduction

In today's fast-paced world, software companies face increasing pressure to rapidly ship new features. Releasing new versions on a daily or weekly basis has become commonplace, even for large, complex software projects [1]. For many years, the major bottlenecks in software development pipelines have been the build and verification processes [2, 3, 4, 5] and much effort in research and industry has been dedicated to improving build tools [1, 2, 6]. Today's build systems are very different from the systems of just a couple decades ago: build systems used to execute build tasks sequentially, while modern build systems distribute single builds across hundreds of machines in different data centers, reusing previous build results wherever possible. Combining a high degree of parallelism and only rebuilding required elements enables these systems to perform massive builds in mere minutes—a task that would take several hours even just a few years ago.

Prominent examples of modern build systems at large companies include Microsoft's CloudBuild [7], Google's Bazel [8], and Facebook's Buck [9], while smaller open source projects often rely on more lightweight but similarly sophisticated build systems, such as Pants [10] and FastBuild [11].

Build tools are a single component of the much larger and more complex software development process [1, 2], so to gain any meaningful speed gains, it is not enough to only focus on tool performance. Cached and distributed build systems (CDBS) can only improve the overall build time by a fraction of what is possible if the core architecture of the software being built does not allow independent tasks to be built in parallel (e.g., poor modularization, unnecessary dependencies between targets). In addition, the complexity of these systems can introduce more bottlenecks in the development process. While caching and distribution provide huge benefits when functioning correctly, these systems are extremely hard to debug when not working as expected. Every internal failure can easily bring the entire continuous integration and deployment (CI/CD) pipeline to a halt, affecting hundreds of engineers, and potentially costing thousands of dollars [3]. Debugging build issues can also take a long time and require considerable effort [12]; meanwhile, the CI/CD pipeline of a major software product might be broken. Although existing tools can help identify and correct a variety of common build errors/failures [13, 14, 15], debugging build performance regressions or refactoring build architecture requires the engineers performing these tasks to understand the build system, its decision making process, and how potential changes to the underlying software architecture will change the tool's behavior. Telemetry and tooling that can provide insights and knowledge into what happened during a build is essential to help diagnose, debug, and repair build issues [12]. Even basic build analysis tasks can quickly overwhelm humans; in fact, Hilton et al. [12] found that *troubleshooting build failures* and *overly long builds* are two of the greatest challenges faced by developers using CI/CD systems.

In our research, we aim to understand the nature of the challenges that developers face with distributed builds and design a tool that can help humans better comprehend and solve build-related issues. We conducted a field study at a large software company working closely with the build system engineers and their first-party customers.

Two research questions guided our study:

RQ 1: What challenges do developers face using modern cached and distributed build systems?

RQ 2: What tasks require tooling support to help engineers understand, debug, and optimize complex build sessions?

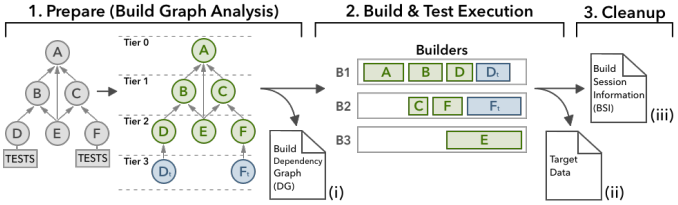

Fig. 1. *CloudBuild* in a nutshell. The three high-level phases that all builds go through: 1) prepare, 2) build and test execution, 3) cleanup.

Although there has been research into reducing build times by optimizing build systems and automatically detecting basic code refactorings [16, 17, 18, 19], little research has been conducted to explicitly address supporting engineers' cognitive needs despite the awareness in industry that cognitive support is a challenge [1, 12]. Although not specifically targeted at CDBSs, researchers have proposed the use of build architecture visualizations [20, 21, 22] and build code change detection tools [13, 23] to cognitively support developers performing complex build-related tasks.

Throughout our research, we found that large software projects struggle to take full advantage of distributed build systems due to these challenges and their own underlying project architecture. Since the problem of understanding cognitive support needs for large distributed builds is not well understood, we used a design science methodology, deeply analyzing this real-world problem faced by domain experts and iteratively designing and validating a solution. Our study at *Microsoft* revealed challenges that engineers face in understanding, debugging, and optimizing CDBS sessions, and helped develop a deeper understanding of their practices, needs, problems, and requirements. These insights supported us as we iteratively designed and developed *BuildExplorer*. We validated our proposed solution through usage scenario walkthroughs, by deploying the tool as it was iteratively designed in the field, and through a controlled user study.

II. Background

We provide a brief overview of how modern build systems [7, 8] work by referring to the tool that was the focus of our case study, *CloudBuild* [7].

There are three high-level phases that all builds go through: (1) *build preparation*, (2) *build execution*, and (3) *build cleanup*. When *CloudBuild* receives a build request, it parses the dependency structure and produces a *build dependency graph* (DG) (see Fig. 1.1). The DG is a directed acyclic graph, where nodes represent *build targets* and edges represent dependencies between targets. Thus, the DG defines the order in which build targets have to be executed. Once the DG is constructed, the *coordinator* starts scheduling individual build tasks across different machines and CPUs (see Fig. 1.1). Targets can only be executed if all their dependencies (source nodes of all incoming edges) were successfully executed. Each target is associated with a *dependency tier*, where targets at dependency tier zero are processed first, and targets having a dependency tier n have to wait for at least one target in tier $n-1$ to be processed before they can begin execution.

Once the DG is produced, the *build and test execution* phase of the build process begins (Fig. 1.2). The *coordinator* tries to optimize build times by assigning unblocked build targets to available builders based on their position in the DG, the estimated execution time, and the preparation work required. The wall-clock time spent in the build phase is defined by the longest time to execute a path of dependent targets. This path is called the *LCP (longest critical path)* of the DG—the longest path in the DG with regard to sequential execution time. LCPs can vary dramatically between builds due to caching dynamics and the distributed nature of a build task, which adds additional complexity when trying to understand builds and their performance. Before executing each target, *CloudBuild* checks if the targets have to be executed or if previous results can be reused (retrieved from cache). A previous result can be reused only if the target definition has not changed (e.g., no code changes) and if none of its dependencies have been rebuilt. When all build tasks complete, the *cleanup* phase starts (see Fig. 1.3): machine states are reset to ensure build isolation and final build information is produced.

Understanding build processes (e.g., why targets are executed and not retrieved from cache) is fundamental for analyzing build sessions. Bazel [8] provides a built-in static node-link dependency graph visualization, while build tools such as Maven [24], Ant [25], and SBT [26] rely on user-driven external plug-ins to provide basic hierarchical node-link graphs [27, 28], text-based dependency trees [29], and exportable graph files that can be uploaded to external visualization tools [30, 31, 32, 33]. However, the generated visualizations are mostly static, while external plug-in visualizations lack context. Although not specific to CDBSs, research has begun exploring the use of visualizations to help developers reason about their products' build systems [20, 21, 22].

As CDBSs continue to rise in popularity, we predict that not having proper tooling to visualize full (or even partial) build structures and build session executions will become a major barrier for developers using these complex systems, which we address in our research. The growing body of research and tools on build comprehension shows that the problem is not unique to *Microsoft*.

III. Research Methodology

Our study used a design science methodology [35, 36], which aims for a deeper understanding of often poorly understood problem spaces. Design science uses problem insights to guide the design of solutions to mitigate those problems, and suggests working closely with stakeholders to iteratively design, develop, and evaluate solution artifacts (such as tools or processes). The goal of design science research is not just to capture the nature of the problem being studied and the justification for the specific solution proposed, but also to arrive at knowledge that can be generalized to other settings.

For our study, we aimed to *characterize the problem domain* of the challenges that arise when engineers have to understand, debug, and optimize cached, distributed build systems. We conducted our study at a software company to

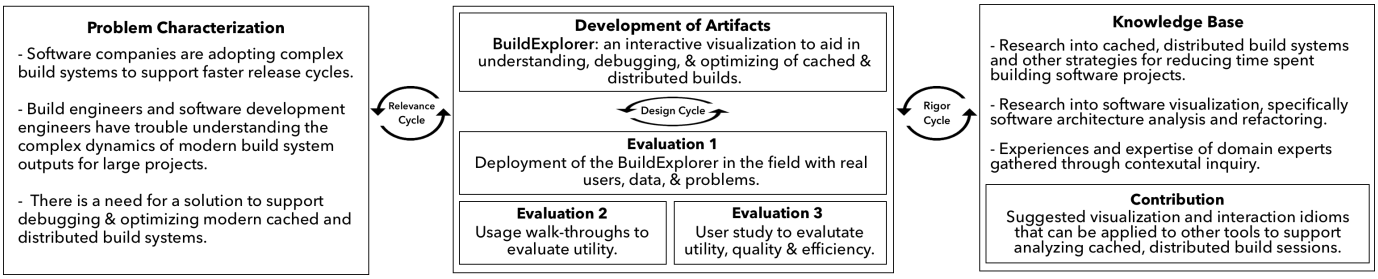

Fig. 2. Overview of our design study and research cycles, based on a diagram by Hevner [34].

inspect a specific instance of the distributed software build problem, and used our gathered insights from working with stakeholders to create actionable design requirements for our proposed *solution artifact*. We iteratively designed and refined our solution (in our case, a tool) by *evaluating* it through deployments in the field, cognitive walkthroughs, and user testing. Finally, we wanted to *reflect* on the requirements we found and our proposed solution so that others may apply this design knowledge to other distributed build systems. Fig. 2 provides an overview of the design study methodology we used to guide our research.

Our study occurred at *Microsoft*, a large international technology company that offers a wide range of products and services. *CloudBuild* is an internal build system that is part of *Microsoft*'s renewed engineering system and is used by many of their products and services. In total, *CloudBuild* executes more than 20,000 builds per day, across more than 500 product branches spanning more than 15 internal organizations.

We worked closely with the engineering team responsible for developing, maintaining, and supporting *CloudBuild*. The broader *CloudBuild* team is divided into 5 sub-teams and counts more than 50 people, including software engineers, product managers, service delivery engineers, and data analysts. In the following sections of the paper, we describe the specific methods we used for gathering requirements (insights on the problem), and designing and evaluating the solution.

IV. Problem Characterization

To answer our two research questions, we used contextual inquiry interviews with the *CloudBuild* team. We identified common challenges developers face when using *CloudBuild*, the main stakeholders that could benefit from improved tool support, and the data they need to help solve their problems. Lastly, we defined an initial set of solution objectives to guide the design and development of *BuildExplorer*.

A. Contextual Inquiry

Contextual inquiry is a semi-structured interview method for gaining valuable contextual information in a highly participatory manner [37]. We conducted interviews with six build engineers who were responsible for developing and maintaining *CloudBuild*, as well as providing support to other engineers using *CloudBuild*. During these semi-structured interviews, participants were asked about the current state of their build session understanding, debugging, and optimization, the data and tooling available, and use cases they felt required further cognitive support.

After the interviews, we observed and continued to question the *CloudBuild* team as they carried out their daily work. Special attention was paid to *CloudBuild* experts that were working on problems related to build optimization and debugging. We also examined the existing tools, build logs, and other artifacts the *CloudBuild* experts were using to help them with these tasks. We explicitly encouraged the *CloudBuild* team to provide suggestions and feedback throughout the artifact design and development processes.

Detailed notes were taken throughout the contextual inquiry process, which took place in person at *Microsoft*. This data was qualitatively analyzed through open coding [38], and the following themes emerged:

(1) **human challenges** when working with build systems;
(2) **potential stakeholders** and their unique requirements;
(3) **data** that could be leveraged to build solutions; and
(4) a set of **solution objectives**.

These themes were later validated with the *CloudBuild* team.

B. Human Challenges

Although previous research has identified some challenges with the CI/CD process as a whole [12], our interviews with build system experts further highlighted the specific difficulties that many users face working with complex modern build systems, such as *CloudBuild*. Although it was clear that engineers value the benefits of these systems, there are also costs. The number of unknown variables and continuously changing states of the cache, build machines, or code under build makes answering simple questions such as "why are builds slower than they were yesterday?" difficult. We identified the following challenges with *CloudBuild*.

Understanding the dynamics of the build system: One main challenge engineers face is understanding the complex dynamics of modern cached and distributed build systems. Debugging build failures requires vast knowledge about the build's dependency graph, the distribution of tasks over machines, and the state of the cache at the time of the build.

The dependency graph of a build is indispensable when trying to understand what happened during a build session. Given unlimited resources, all targets at the same *dependency tier* could hypothetically be built in parallel as they have no dependencies between them. However, sometimes dependency tiers contain thousands of targets, making parallel execution infeasible. In this case, the build system needs to select

execution targets based on a variety of metrics: position in LCP, build duration, number of dependencies, etc. Thus, proper build comprehension also requires detailed knowledge of the decisions the build system made.

End-to-end build time is dependent on the time taken to build the longest-building dependent sequence of targets in the dependency graph, the LCP. However, with caching enabled, unchanged targets can be retrieved from cache, which may result in very different LCPs between builds. To understand why a build took a long time, engineers need to understand the caching dynamics as well as the LCP.

Lack of visibility into why targets are being rebuilt: In extremely large projects, *CloudBuild* users sometimes have a hard time determining why specific targets are rebuilt. Often, users that are unaware of target dependencies are confused when downstream targets are rebuilt. In legacy build systems, engineers could rely on idempotent build sessions—building the same set of code changes on different machines would result in the same targets being executed— but this is often not the case with CDBS like *CloudBuild*.

Builds can have dramatically different LCPs and build times (even when building the same code change twice in a row), adding complexity when trying to understand builds and their corresponding run times. Although the state of the cache is hard to capture and comprehend, this information is needed when trying to understand why a target had to be rebuilt. Cache misses can require the re-execution of thousands of dependent targets, causing delays in the CI/CD pipeline and forcing other builds to queue due to occupied resources.

Data fragmentation: Another commonly mentioned challenge was the scattering of build data across many locations (logs, databases, tools, systems), while different data stores used diverse and often incompatible data schemas. This makes it extremely difficult for engineers to find all relevant build session information, let alone draw insights from this data. Engineers had to rely on multiple tools to explore the data emitted by *CloudBuild*, most of which provided inadequate support. They often had to use raw log files or query telemetry databases to try to find what they needed, repeatedly wishing for a standalone tool, in the form of an easy-to-access web visualization, that could present a holistic view of a build.

C. Stakeholder Characteristics

We identified three user groups that would benefit from a dependency graph visualization tool to ease build session understanding and to support build system debugging and optimization tasks. Fig. 3 depicts these user groups and their common interaction patterns. These user groups are not mutually exclusive and may have overlapping build tool needs.

(1) Build system engineers have expert knowledge of the internal workings of the build system, and actively *develop* and *maintain* the build service. Although they have detailed knowledge of the data that *CloudBuild* generates, they often have trouble navigating the complex set of logs required to analyze build failures. Build system engineers are exposed to build issues and failures on a constant basis. They onboard new

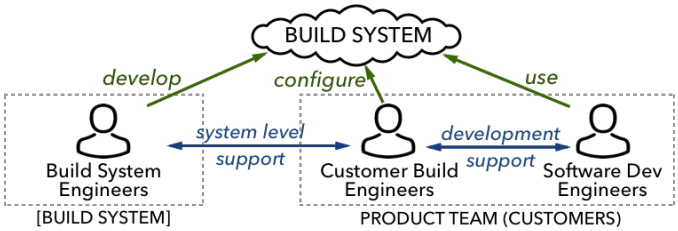

Fig. 3. The three categories of *CloudBuild* stakeholders.

users and support customers trying to leverage the distributed and cached nature of *CloudBuild*. Build system engineers are often the first point of contact when *CloudBuild* users have questions, working 24/7 to address their issues as quickly as possible. As build tools are essential in many development pipelines, any build-related issues could potentially threaten a team's ability to deliver software on time. Thus, the time taken to resolve these issues is crucial.

(2) Customer build engineers work on product teams that *use CloudBuild* in their engineering workflows. They are responsible for *configuring* builds and *maintaining* build structures for their team. This user group serves as the main interface to the *CloudBuild* engineers, and manage the integration of the build system into their product's development pipelines, usually in partnership with the *build system engineers*. While performance regressions and reliability of the build system itself are the responsibility of the *build system engineers* (see above), *customer build engineers* are responsible for monitoring the overall status of their team's builds. They quickly need to identify and understand any code changes that negatively impact the build (e.g., adding a new dependency between two components) so that they can take action (e.g., reject the change, trigger a refactoring). They also want powerful visuals to help them better understand and share the impact of code changes with the rest of their team.

(3) Software development engineers work on product teams and *use* the build system in their daily development workflows. This group often lacks detailed knowledge about how the build system works and the impact that their code changes may have on the overall development pipeline. The main concern of this user group is the end-to-end build time (i.e., how long they may have to wait for a build result). If build time dramatically deviates from what they expect, they may start a high-level investigation into what happened. However, if they fail to find the cause, they usually seek the assistance of one of their team's *customer build engineers*.

D. Data Characteristics

Through our discussions with *build system engineers* and an examination of their telemetry data and support tools, we identified three main categories of *CloudBuild* build data.

Build session information (BSI) contains high-level metadata about a build session. The BSI object summarizes the build configuration and request options, which are essential for understanding the context of the build. The BSI object also provides a summary with statistics of the build results,

such as time spent in each build phase, resources used, build results, and potential errors.

Build dependency graphs represent the dependencies between build targets at the time of the build. As discussed earlier, the dependency graph provides insights into the overall connectivity of the project and defines a lower bound of how fast any build system can build the underlying software system. The dependency graph is crucial for understanding the build system's choices, e.g., the target build order.

Target information contains details about each target's execution, such as if the target was retrieved from cache, how much memory was consumed during execution, the amount of processing time, the size of the inputs and outputs, the type of the target, and the name of the machine that processed the targets. This information is essential for understanding what happened to the individual targets during a build session.

E. Solution Objectives

As expected, many of the tool support needs for the three user groups overlap. However, for the initial design of the *BuildExplorer* tool, we focused on the needs of *build system engineers* as they have expert knowledge and are also users of the build system. We identified two main use cases:
Use case 1: Debugging a build session for specific performance issues. This task requires deep analysis of diverse data.
Use case 2: Optimizing the performance of end-to-end builds for customer projects. This task often leads to recommendations for software architecture refactoring.

Based on these two main use cases, we identified a set of requirements that a build visualization tool should satisfy. We iterated and validated these requirements through discussions with the *build system engineers* until we arrived at a relatively stable set of tasks to be supported by our *BuildExplorer* tool:

Task 1: showing high-level **build session information**;
Task 2: visualizing the **overall structure** and **distribution** of dependencies;
Task 3: providing insights where **build time was spent**;
Task 4: highlighting the build's **caching dynamics**; and
Task 5: allowing one to browse **build target information**.

Using these requirements, we designed an interactive tool that visualizes a build's dependency graph while also allowing the user to drill into other build-related details.

V. The BuildExplorer Solution

BuildExplorer was designed to help *build system engineers* understand build graph structure, analyze build performance, and diagnose possible build-related issues. We worked closely with the *CloudBuild* team to generate and evaluate multiple ideas and ensure we produced the best solution, we developed low-fidelity mockups and incorporated the *build system engineers'* feedback. To allow for easy access and integration into the existing *CloudBuild* ecosystem, we implemented the selected solution as an ASP.NET web application, complemented with D3.js [39] and HTML canvas [40]. In the following section, we describe the design and implementation details along with the rationale behind our choices, and include a brief discussion of discarded alternatives where appropriate. A full demo of *BuildExplorer* is also available online [41].

A. Layout and Interactions

The *BuildExplorer* tool provides a series of coordinated views, linked by visual and interactive strategies that showcase different aspects of the build data, as shown in Fig. 4. These coordinated views allow users to actively explore the complexities of build session data from a variety of perspectives, promoting deeper understanding and better decisions [42].

The views were designed to fit on a single screen to reduce cognitive overhead from switching and navigating between screens. Our goal was to provide a highly interactive user experience with fast overviews and easy filtering to support a details-on-demand analysis [43]. A *search bar* allows users to quickly find and filter targets of interest (Fig. 4(i)). *Hovering* is used throughout the tool to provide extra information and link data items across views, e.g., hovering over a target triggers a tool-tip summarizing the target's key properties and highlights the target across all coordinated visualizations. Clicking on targets *selects* them in the connected visualizations and highlights their direct dependencies.

B. Views and Visualizations

BuildExplorer's user interface features two summary views and a set of coordinated visualizations. We use (X) to refer to the corresponding view/visualization in Fig. 4.

(A) **The Build Header View** consolidates build information (e.g., parameters, settings) to provide contextual information that impacts the behavior of a build in an easy to understand visualization. This information is critical for nearly all debugging scenarios. Since presenting all the build settings would overload the view, we worked with *CloudBuild* engineers to determine what was essential (e.g., build type, architecture, caching, change id) and provided a link to the complete logs.

(B) **The Build Summary View** provides a visual breakdown of the stages and caching dynamics of a build session. The *build timeline* visualization (Fig. 4(ii)) shows the time spent in each build stage (*Task 3* from Sec. IV-E), e.g., preparation, build, cleanup, etc. The *target cache* visualizations (Fig. 4(iii)) provide cache statistics per build target type (*Task 4* from Sec. IV-E). Combined, the two build summary components present a high-level overview of the dynamics of a build session (*Task 1* from Sec. IV-E).

(C) **The Dependency Graph Visualization** provides an overview of the project's dependency structure. To help users understand and navigate their complex code bases, many existing tools leverage common graph visualization techniques, such as hierarchy visualizations [44, 45, 46], nested tree visualizations [47], node-link diagrams [20, 21, 45, 48], and hybrid visualizations [49, 50]. However, these existing graph-based approaches struggle to scale to larger software projects [20].

We used rapid prototyping to visualize the dependency graph as a whole (or even in parts) using traditional network graph techniques. However, due to the size and density of even smaller build dependency graphs processed in *CloudBuild*,

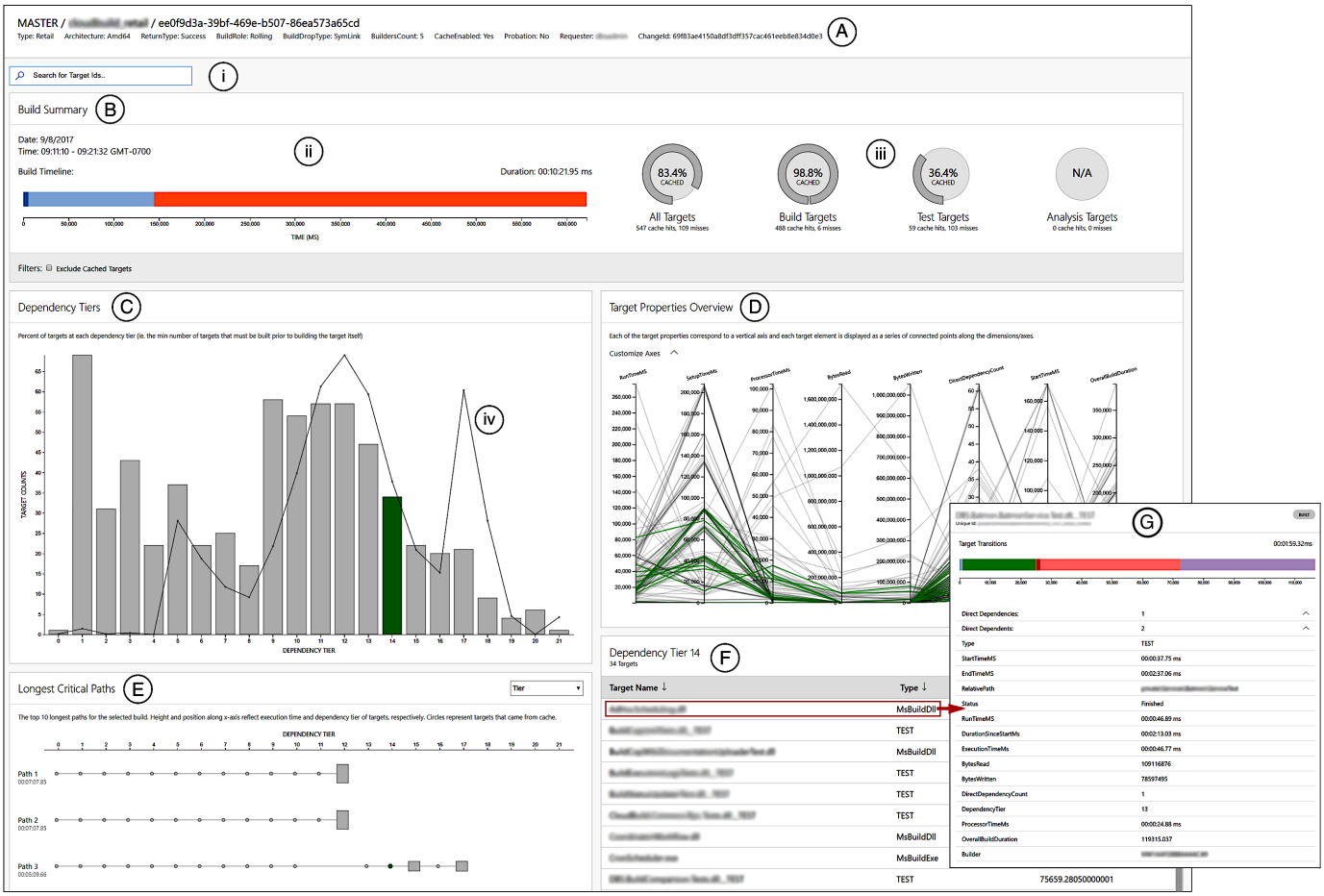

Fig. 4. The proposed *BuildExplorer* solution, featuring: (A) Build Header View, (B) Build Summary View, (C) Dependency Tiers Visualization, (D) Target Properties Visualization, (E) Longest Critical Paths View, (F) Target List View, (G) Target Details View.

these early attempts were almost illegible. Automatic layout strategies generated tangled *"hairballs"* [51] that provided users with little actionable information. Graph reduction techniques [48, 52, 53, 54] hid important relationships between targets, while expanding nodes left the graphs cluttered. Although researchers have shown promising results using query languages and filtering mechanisms to help reduce and simplify their build graph visualizations [20], we ruled out these and other common filtering strategies [43] as the different usage scenarios required complex filtering options which quickly became cumbersome and did not scale to massive software projects (≈16,000 targets, ≈20,000 dependencies).

Instead, we were inspired by an existing, internal visualization that uses a histogram to group build targets according to their dependency tier and colors them based on their current status (e.g., in process, finished). However, this visualization has its own set of weaknesses: targets are hard to find, the amount of color-coding is overwhelming, and when the tier "stacks" become too tall to fit on the screen, they overflow into a new stack.

Consequently, our *dependency tier* visualization is an updated and refined version of the original histogram. It visualizes the overall structure and distribution of targets in the build's dependency graph (*Task 2* from Sec. IV-E). Each bar in the histogram represents the number of targets (y-axis) in the corresponding dependency tier level (x-axis). The distribution of targets across tiers provides an estimate of where targets are concentrated in the build graph and whether there are any funnels or bottlenecks in the graph. With this visualization, users can get a sense of the overall structure and health of the build dependency graph, as discussed in Sec. VI-A.

Overlaid on the *dependency tier* visualization is the **tier timeline** (Fig. 4(iv)), which provides a high-level overview of the relative distribution of time spent across all dependency tiers—the sum of target execution or target cache fetch times overall targets for each dependency tier. Although we grouped targets by dependency tiers in this visualization to meet the *build system engineers'* needs, this style of visualization could easily be adapted to suit other stakeholders' needs and future use cases (e.g., to show machine utilization or target build start-time bins).

(D) **The Target Properties Visualization** provides an easy comparison across many targets and their relationships. Build targets are visualized using a parallel coordinates plot [55], where each of the target's properties (e.g., execution time, start time, bytes read or written) are represented as parallel, vertical axes. Target values are plotted as a series of connected lines across all axes (*Task 5* from Sec. IV-E). The arrangement

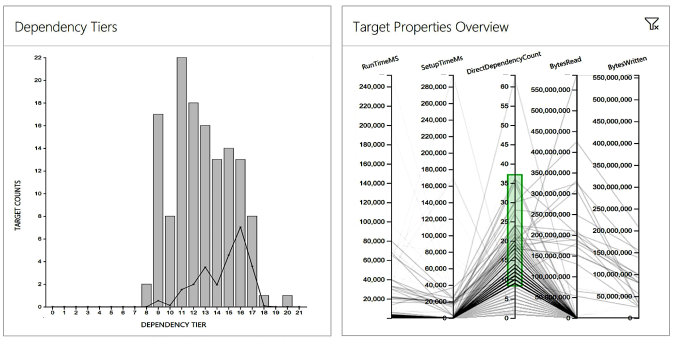

Fig. 5. Brushing technique used to select targets with a particular range of property values. The dependency tier view gets filtered upon selection.

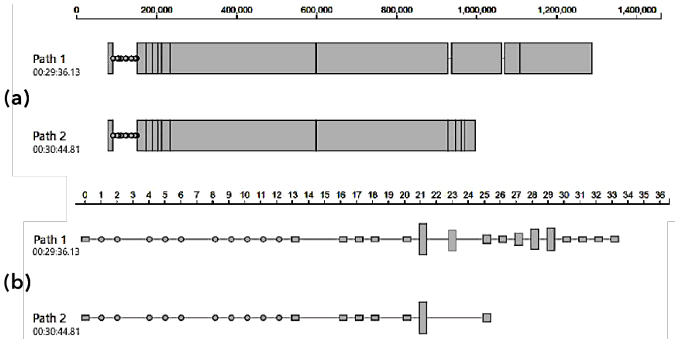

Fig. 6. The two versions of the Longest Critical Path visualization: (a) the default tier-based visualization, and (b) the time-based visualization. The circles and bars reflect targets coming from cached and built targets, respectively. The size of the bar represents the target's build time.

of lines between axes shows possible relationships: roughly parallel lines suggest a *positive relationship*; crisscrossing lines (x-shape) suggest a *negative* or *inverse relationship*; while randomly crossing lines suggest *no relationship*. The axes can be reordered, added, or removed to examine relationships between various target properties.

To avoid clutter, users can brush [56] along axes to select targets within ranges, fade out unselected targets, and *filter* the data in the connected visualizations. Individual targets (or groups of targets) in the *target properties* visualization can also be selected by directly clicking on them.

(E) **The Longest Critical Paths Visualization** shows a list of the top 10 longest critical paths (LCPs). Regardless of the number of resources, a build's end-to-end execution time is bounded by the execution time of the LCP. Targets situated on the same LCPs are drawn as a series of connected bars (built targets) or circles (cached targets). The LCP's targets can be arranged along the horizontal axis, either based on their dependency tier or start time. In the **tier-based LCP** visualization, targets are horizontally aligned with their dependency tiers, while the height represents target execution time (Fig. 6(a)). Using the **time-based LCP** arrangement, targets are horizontally aligned by their start and end times, while the width of a box represents execution duration (Fig. 6(b)). This visualization supports *Tasks 3 and 4* from Sec. IV-E.

(F) **The Target List Visualization** is visible upon initial load or when a set of targets have been selected. Users can sort the list of targets by properties (e.g., name, build time).

(G) **The Target Details Visualization** is only shown when a single target is selected and provides information regarding what happened to that target during the build session (*Task 5* from Sec. IV-E). When a target is being executed (or retrieved from cache), it goes through a series of phases called target transitions. The *target transition timeline* bar graph visualization shows the time a target spent in each transition phase (*Task 3* from Sec. IV-E). Underneath the timeline is a list of the target's direct dependents and dependencies, as well as a list of key target properties.

VI. Evaluation

We used three different methods (congruent with Venable *et al.'s* framework for evaluating design science research [57]) to evaluate the *utility*, *quality*, and *efficiency* of our solution:

a) two *usage scenario walkthroughs* to demonstrate how the tool satisfies the *utility* requirements for the two use cases identified in Sec. IV;
b) ongoing *field deployments* of the solution to validate the *utility* requirements and to *shape the tool's design* in a natural setting; and
c) a *user study* in a controlled, artificial setting to consider the *utility, quality, and efficiency* of the solution.

A. Usage Scenario Walkthroughs

We outline how the set of tasks the tool was designed to support can help *CloudBuild* engineers in two scenarios.

Scenario 1 - Build Optimization: Here, the build engineer is contacted by a customer on-boarding to *CloudBuild*. The customer wants to know if they can speed up their builds. The build engineer, recognizing the customer is new to the build system, examines the *dependency tier* view to determine the overall structure and parallelization potential of the project's current build dependency graph. Fig. 7 shows the *dependency tier* visualization for three sample plausible builds:

(a) A *healthy* dependency graph. The tiers have a positively skewed distribution, indicating that most of the targets can immediately be built in parallel. This build fully utilizes the distributed nature of the CDBS.
(b) A potentially *unhealthy* dependency graph. The tiers have a negatively skewed distribution, indicating that most of the targets have to wait for a series of dependencies to build. This could possibly be slowing down the overall build time since the beginning of the build does not leverage the distributed nature of the CDBS.
(c) A *bottlenecked* dependency graph. This build has many targets waiting for the outputs of just a few targets.

With this view, the build engineer may spot possible performance optimizations that will decrease build times, e.g., refactoring the dependency structure to allow for greater parallelization of build tasks. The *tier timeline* graph may be consulted to determine how much time is spent in each dependency tier. If the dependency tier graph is negatively

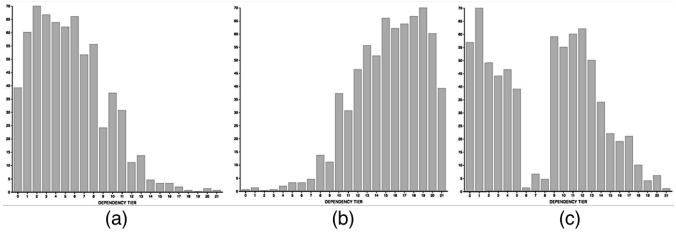

Fig. 7. Three vastly different dependency graph structures visualized with the *dependency tier* view: (a) healthy positively skewed graph, (b) unhealthy negatively skewed graph, (c) unhealthy bottlenecked graph (tiers 6 - 8).

skewed but spends relatively little time building targets in the larger tiers, the build may still result in a good run time.

Since the lower bound of a build's run time is determined by its LCP, the next step in identifying potential causes for slow builds is the *LCP* visualization. The build engineer can check the *tier-based LCP* visualization to identify targets in the LCP that might be bottlenecks, e.g., have a high dependency tier and therefore must wait for the outputs of many targets. They can switch to the *time-based LCP* visualization to identify expensive (long-running) build targets.

Lastly, the build engineer can explore the *target properties* view to filter for abnormalities, e.g., long-running or highly dependent targets. If these targets (or any of their dependencies) change frequently, it can become very costly to the overall build and development process. Possible mitigations include splitting these targets or removing unnecessary dependencies.

Scenario 2 - Build Debugging: Here, the build system engineer is contacted by one of the build system's end-users (likely a Customer Build Engineer) who is worried about their project's build times, e.g., the builds are suddenly unusually slow. The customer wants to understand what changed and the implications of those changes on overall build performance. The first step is to check if the customer's complaints are justified and there is actually a build regression. The *build summary* view can then be used to check for long queue or setup times, indicating missing resources, or dramatic changes to *target caching* dynamics that could explain the slowdown.

Once a performance issue has been confirmed, the build system engineer can check the *LCP visualization* to see if any targets are taking an abnormally long amount of time, or if any new targets have been added. The build system engineer can also check the overall health of the build's dependency graph to see if new bottlenecks were introduced, if the general distribution of tiers changed, or if new tiers were added. Lastly, they can inspect the *target properties* to identify targets whose build times have increased significantly, or to check if any new long-running targets were added.

B. Deployments in the Field

Since the development of *BuildExplorer* was iterative and incremental, we made our prototypes and early versions of the tool available as soon as possible to observe its use with real users. We encouraged users to provide feedback and to share ideas for future features. This early feedback shaped the development of the tool, validated the problem characterization, and helped us refine the solution objectives (see Sec. IV). We found that the features users often requested were consistent with our initial development roadmap, which further validated our design choices and user tasks. Below, we highlight some key experiences to illustrate the nature of the feedback we received.

Tasked with helping a new team on-board to *CloudBuild*, one early adopter used *BuildExplorer* to optimize the product's average build time. They used the *dependency tier visualization* to understand the project's architecture, identify candidates for dependency refactorings, and determine the project's overall parallelization potential. With the help of *BuildExplorer*, the onboarding team was able to re-shape the graph from an *unhealthy*, negatively skewed distribution to a *healthier*, positively skewed distribution. The *target list* and *target properties* also identify targets with high build times and identified potential target refactorings. Following these recommendations, the team reported large speed improvements for both cached and non-cached builds. Another reported experience was exploring possible causes of a suspected build performance regression, brought to the *CloudBuild* team's attention by a customer. Using the *LCP* and *target details* views, they were able to identify a stalling build machine and sent screenshots of their findings to a co-worker for follow-up.

C. User Study

Since the type and frequency of real-life incidents (such as the ones described above) vary dramatically, we designed a study to simulate a common usage scenario that would require the participants to use a variety of *BuildExplorer's* views/visualizations, allowing us to better gauge the target user's perceptions of the tool.

1) Methodology: We used purposive sampling to recruit five *CloudBuild* participants. To ensure participants were representative of **build system engineers**, they had to hold a technical role and have experience working with *CloudBuild*. Four sessions were conducted in person and one was conducted through a video conferencing tool.

All participants had actively worked on the *CloudBuild* code base, ranging from a few months to six years. Four of the five participants described themselves as having extensive experience debugging *CloudBuild* performance issues, spending an average of 6-10 hours weekly on these tasks. Prior to the user testing session, two participants had briefly used *BuildExplorer* and the other three had seen a tool demo. After participants granted consent for their participation, they briefly explored the tool and received a tour of the tool showing them details they would normally find in documentation or online tutorials.

2) Task Description: Each participant had 30 minutes to complete the same short task using a think-aloud protocol [58]: *"A customer contacts you about one of their build queues. They've noticed that some of their builds have been taking longer than normal and want you to help them understand what has changed. Try using the BuildExplorer tool to investigate the builds so you can report back to the customer."*

During our contextual inquiry, we found that *CloudBuild*'s customers typically sent a few examples when they reported build regressions, e.g., normal builds and builds behaving unexpectedly. Based on this, the study participants were provided with four builds: one *fast* and three *slow* builds. We selected builds from one of the largest projects using the *CloudBuild* service (≈16,000 targets and ≈20,000 direct dependencies). Since the participants actively debug performance-related issues in *CloudBuild*, we selected three *slow* builds with no known issues or associated tickets. We inspected a variety of slow builds with *BuildExplorer* and chose a build with very low cache rate, and two builds that exposed abnormal build and cache retrieval behaviors. Detecting the last two build issues would require thorough build log inspections.

3) Findings & Insights: After the initial tour of the tool, we found that participants were able to use *BuildExplorer* with little additional guidance from the facilitators–only asking the facilitator to repeat information from the tutorial or the facilitator prompting a stalled participant by asking if they had looked at *x* visualization. During the experiment, participants vocalized a variety of hypotheses as to why they believed the builds were slower than the normal build, e.g., low cache-hit rates, too few build resources, etc. These hypotheses stemmed from their interactions with the tool and provided a starting point for further investigations. After completing the task, participants were asked to reflect on their experiences. All sessions were video-recorded and analyzed for common themes. Transcriptions were made of the post-task interviews, and through qualitative analysis, several themes emerged.

The overall response to the user testing session was very positive. One participant commented, *"I think this is awesome!"* and later added, *"I'm like a kid in a candy store—there are so many cool things about it"*. All participants agreed that they were very likely to use the tool to support their duties as *CloudBuild* engineers. All participants also suggested additional features and visualizations they would like to see added to the tool, as summarized in Sec. VIII.

Below we discuss some key observations and feedback gathered during the user testing sessions, specifically reflecting on the *efficiency*, *utility*, and *quality* of the tool.

(Efficiency) Consolidating views of the data: Designing a single screen for all views was seen as the right approach as it combined many different data perspectives and reduced navigation needs, as one participant noted: *"The fact that it's able to condense a lot of information in one area is really useful. The density of information without [it] being overwhelming is very valuable."* Another participant noted, *"I don't have to go to multiple locations—I can get it all in one."*

(Utility) Understanding overall build processes: Surprisingly, *CloudBuild* engineers felt their own build system is hard to understand and that even for experts, *BuildExplorer* is *"really good to understand the overall build process"*. *"It also permits you to dig deeper to understand the build structure, the actual build happening, and well as hypothetical limits that you can achieve for various performance related scenarios"*.

(Utility) Starting point for investigations: *BuildExplorer* was never meant to be a completely comprehensive tool for debugging all *CloudBuild* issues. However, it is clear that *BuildExplorer* will remain a starting point for many debugging scenarios, including customer questions: *"We get a lot of questions on [internal Q/A site] like, 'my build was slow'. This is a good way to go investigate that."* One participant stressed that there will always be a need for build logs in the engineering process and we absolutely agree. *"During the investigation, I'm going to need to look at logs, no matter how much information you put here. As an engineer, that is just always going to happen [in] the last mile of investigation, once we know something—to prove it!"* While *BuildExplorer* may allow users to understand the scope and potential source of problems, deeper investigations will likely require many different tools and data sources.

(Utility) Longest critical path: The LCP view was extremely popular as it allows fast detection of many build issues. For example, *"the longest critical paths were the most useful to understanding what exactly happened,"* and *"the dependency tiers [are] very helpful, and so [are] the longest critical paths."* Another participant pointed out that the tier-based LCP is better for understanding *"hypothetically what you could do to improve"*, whereas the time-based LCP allows exploring *"what happened in this [build] or what went wrong"*.

(Quality) Intuitive and interactive: We observed that users, for the most part, were able to navigate the tool with ease. The participants described the tool as having a *"good, intuitive UI"* and being *"easy to figure out"*. There was excitement due to the interactive nature of the tool. Users *"really liked that it's all connected"* and enjoyed that selections *"let you focus on that thing in the other views"*.

(Quality) Tool Complexity: Ideally users should understand how to use a tool without the need for any explanation or supplementary material, but for complex tasks, it is likely any tool will also be complex. Even though the participants were CDBS experts, some users were unsure of some of the data shown and the data mappings. Most of the confusion stemmed from the target properties view (Fig. 4h), a visualization type that none of the participants had previously seen or used. When reflecting on using the tool, one participant commented that *"[I] was completely lost about the target properties overview... I wasn't able to utilize it"*. *"I don't really know how to exploit this"*, mentioned another participant about the target properties view. However, when given a more detailed demonstration of the target properties view, one participant responded, *"oh, I didn't know about that...okay, yeah that's very useful!"*, which may indicate that we did not provide sufficient training or exploring time before starting the task. Although the participants had briefly seen the tool before, this was the first time the participants were using it on their own.

VII. Discussion

To understand the challenges developers face with CDBS and develop a solution to support them, we conducted a design study [34, 35, 36] at *Microsoft*. Our design science methodology and key findings are summarized in Fig. 2.

To ensure the problem we addressed was *relevant* (*relevance cycle* Fig. 2 [34]), we studied a particular instance of the problem working closely with domain experts at *Microsoft*. Through *BuildExplorer*, we further ensured that the proposed solution addressed a relevant problem and that our tool would extend the current capabilities of our target users.

To ensure *rigor* (*rigor cycle* in Fig. 2 [34]), we designed our solution using existing knowledge from the target domains: build systems, CI/CD optimization, and software visualization research on architectural analysis and optimization.

To extend the *knowledge base* and our contribution to research, we propose visualization and interaction idioms that can be applied to similar tools striving to support human understanding of CDBS. Although previous work has explored strategies for supporting cognitive needs of the build engineers when performing basic debugging and optimization tasks, these strategies have both failed to scale to the size of modern software projects and do not explore the complexities of non-deterministic CDBS sessions. Although our study was conducted on one build tool at a single company, many modern build systems share similar features. However, validating that this knowledge applies to other tools remains future work.

Lastly, through our *design cycle* [34], we evaluated the *utility*, *quality*, and *efficiency* of *BuildExplorer* solution through usage scenario walkthroughs, deployments in the field, and a user study. In this cycle, we also considered alternative solutions and justified our final solution.

A. Lessons Learned

Here, we share some key lessons learned that may be of general interest to practitioners and researchers.

Take time to understand usage scenarios and preliminary tool designs in the wild: Since the CDBS system problem space was largely unexplored, investing the time to understand and characterize the domain prior to and during the design of a tool was crucial to the tool's success. Insights into the usage scenarios and needs only became clear as people started using the tool in their workflows. Once the users' basic needs were partially satisfied through early iterations of the tool (as one participant pointed out, it's *"better than anything we have at this point"*), there was a flow of new ideas about how to refine existing and design new features. The design science methodology helped us structure the problem space and extract key insights that helped in the design and evaluation of the proposed solution (see Fig. 2).

A complex solution for a complex problem requires training: The majority of the feedback about the tool was positive—any tool support was seen as better than no support. While the single view reduced navigational overhead, we noticed that some users became overwhelmed with the amount of data shown. It took time for users to understand what was being shown and the benefits of linking information across views. Other times, the users required additional explanation from the tool designers before they felt comfortable using the tool. Thus, with such a complex tool, thorough training and documentation should be provided, and users should be encouraged to take the time needed to fully understand and feel comfortable using *BuildExplorer*.

Build logs are still essential: Just as developers report a lack of trust in refactoring tools [59, 60, 61], it is not surprising that they also prefer to directly refer to the build logs when confirming their hypotheses concerning performance issues. *BuildExplorer* supports this preferred work-flow by providing users with a starting point for build analysis and links to the required build logs so they can continue their investigations. Linking to build logs within *BuildExplorer* also has the added benefit of helping to address the data fragmentation issue identified during our contextual inquiry.

B. Research Limitations

This research was conducted at one organization using a single build system. Our findings may not generalize to other organizations and build systems. To ensure broad relevance, we identified key stakeholders, use cases, and tasks which we believe need support regardless of the underlying build system.

We also acknowledge that the low number of participants in our user study limits the generalizability of our research. We relied on *CloudBuild* experts as this was our primary target audience, and so we cannot currently generalize our findings to other user groups (see Sec. IV-C).

Due to our data collection methods, we may also have unintentionally introduced researcher bias. Since we conducted an in-person study, we may have introduced *interviewer bias* through subtle clues (e.g., body language, tone) that may have influenced the participants' responses or actions. Conversely, the participants may have consciously, or subconsciously, provided responses they believed we wanted to hear (e.g., positive feedback on the tool in the user study), introducing *response bias*. However, we attempted to mitigate this by encouraging participants to discuss areas for improvement.

VIII. Future Work and Conclusion

Although *BuildExplorer* was eagerly adopted at *Microsoft* and has received positive feedback, there are many areas for improvement. The current version of the tool focuses on *CloudBuild* engineers and covers a small subset of the daily occurring scenarios. To support more scenarios or user groups, many views and additional data need to be added, e.g., historical data, comparing builds, advanced filtering. Finally, understanding the impact of *BuildExplorer* on software architecture and distributed building practices over time remains future work.

Although there remains much work to be done in this domain, we believe that the work presented in this paper is a good initial step towards understanding, debugging, and analyzing build performance. *BuildExplorer* has tremendous potential to help other organizations that use similar cached and distributed build systems.

Acknowledgments

We thank the members of the TSE and CloudBuild team for their advice and support. Special thanks to Yash Upadhyay, Steve Culver, Jacek Czerwonka, and Cassandra Petrachenko.

References

[1] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, March 2016, pp. 78–90.

[2] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[3] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 483–493.

[4] B. Murphy, "Optimizing software development processes," in *2016 IEEE/ACM 4th International Workshop on Conducting Empirical Studies in Industry (CESI)*, May 2016, pp. 4–4.

[5] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *Proceedings of the Third International Workshop on Managing Technical Debt*, ser. MTD '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1–6.

[6] G. Maudoux and K. Mens, "Correct, efficient, and tailored: The future of build systems," *IEEE Software*, vol. 35, no. 2, pp. 32–37, March 2018.

[7] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "Cloudbuild: Microsoft's distributed and caching build service," in *ICSE SEIP*. ACM, June 2016.

[8] "Google bazel." [Online]. Available: http://www.bazel.io/

[9] "Facebook buck." [Online]. Available: https://buckbuild.com/

[10] "Pants." [Online]. Available: https://www.pantsbuild.org

[11] "Fastbuild." [Online]. Available: http://fastbuild.org

[12] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 197–207.

[13] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for make-based build crashes," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 526–530.

[14] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 106–117.

[15] F. Hassan and X. Wang, "Hirebuild: an automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1078–1089.

[16] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Identifying and understanding header file hotspots in c/c++ build processes," *Automated Software Engineering*, vol. 23, no. 4, pp. 619–647, 2016.

[17] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos, "Improving the build architecture of legacy c/c++ software systems," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 96–110.

[18] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing false code dependencies to speedup software build processes," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '03. IBM Press, 2003, pp. 343–352.

[19] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 123–133.

[20] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 114–123.

[21] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 650–660.

[22] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 398.

[23] C. Macho, S. McIntosh, and M. Pinzger, "Extracting build changes with builddiff," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 368–378.

[24] "Apache maven," 2017. [Online]. Available: https://maven.apache.org

[25] "Apache ant." [Online]. Available: http://ant.apache.org/

[26] "Scala sbt." [Online]. Available: http://www.scala-sbt.org/

[27] "Depgraph maven plugin." [Online]. Available: https://github.com/ferstl/depgraph-maven-plugin

[28] "Maven graph plugin." [Online]. Available: https://github.com/janssk1/maven-graph-plugin

[29] "Visualizing project dependencies in sbt." [Online]. Available: http://xerial.org/blog/2014/03/27/visualizing-project-dependencies-in-sbt

[30] "Ant grand." [Online]. Available: https://ant-grand.github.io/

[31] "Vizant." [Online]. Available: http://vizant.sourceforge.net/

[32] "Ant2dot." [Online]. Available: http://ant2dot.sourceforge.net/

[33] "Sbt dependency graph." [Online]. Available: https://github.com/jrudolph/sbt-dependency-graph

[34] A. R. Hevner, "A three cycle view of design science

research," *Scandinavian journal of information systems*, vol. 19, no. 2, p. 4, 2007.

[35] A. Hevner, A. R, S. March, S. T, P. , J. Park, R. , and S. , "Design science in information systems research," vol. 28, pp. 75–, 03 2004.

[36] M. Sedlmair, M. Meyer, and T. Munzner, "Design study methodology: Reflections from the trenches and the stacks," *IEEE transactions on visualization and computer graphics*, vol. 18, no. 12, pp. 2431–2440, 2012.

[37] K. Holtzblatt and S. Jones, "Contextual inquiry: A participatory technique for system design," *Participatory design: Principles and practices*, pp. 177–210, 1993.

[38] K. Charmaz, *Constructing grounded theory*. Sage, 2014.

[39] M. Bostock. (2017) Data-driven documents. [Online]. Available: https://d3js.org/

[40] "Mdn web docs: Canvas." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas

[41] "Cloudbuild graph explorer," Microsoft. [Online]. Available: https://www.microsoft.com/en-us/research/project/cloudbuild-graph-explorer/

[42] J. S. Yi, Y. ah Kang, and J. Stasko, "Toward a deeper understanding of the role of interaction in information visualization," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, pp. 1224–1231, 2007.

[43] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proc. of Symp. on Visual Languages*. IEEE, 1996, pp. 336–343.

[44] M.-A. Storey and C. Callendar. (2017) Creole. [Online]. Available: http://www.thechiselgroup.org/creole/

[45] "Directed graph markup language (dgml) reference." [Online]. Available: https://msdn.microsoft.com/en-us/library/dn966108.aspx

[46] M. Lanza, "Codecrawler: polymetric views in action," in *Proc. of Int. Conf. on Automated Software Eng.*, 2004, pp. 394–395.

[47] P. O. Kristensson and C. L. Lam, "Aiding programmers using lightweight integrated code visualization," in *Proc. of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2015. New York, NY, USA: ACM, 2015, pp. 17–24.

[48] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[49] M. Pinzger, K. Graefenhain, P. Knab, and H. Gall, "A tool for visual understanding of source code dependencies," in *the ICPC*. IEEE, 2008, pp. 254–259.

[50] J.-M. Maatta, M. Honkonen, T. Korhonen, E. Salminen, and T. Hamalainen, "Dependency analysis and visualization tool for kactus2 ip-xact design framework," in *Int. Symp. on System on Chip*. IEEE, 2013, pp. 1–6.

[51] R. KOSARA. (2012) Graphs beyond the hairball. Eager Eyes. [Online]. Available: https://eagereyes.org/techniques/graphs-hairball

[52] E. R. Gansner, Y. Hu, S. North, and C. Scheidegger, "Multilevel agglomerative edge bundling for visualizing large graphs," in *Pacific Visualization Symp. (PacificVis), 2011 IEEE*. IEEE, 2011, pp. 187–194.

[53] D. Holten and J. J. Van Wijk, "Force-directed edge bundling for graph visualization," in *Computer graphics forum*, vol. 28, no. 3. Wiley Online, 2009, pp. 983–990.

[54] W. Sadiq and M. Orlowska, "Analyzing process models using graph reduction techniques," *The Int. Conf. on Advanced Information System Engineering*, vol. 25, no. 2, pp. 117 – 134, 2000.

[55] A. Inselberg and B. Dimsdale, "Parallel coordinates for visualizing multi-dimensional geometry," in *Computer Graphics 1987*, T. L. Kunii, Ed. Tokyo: Springer Japan, 1987, pp. 25–44.

[56] R. A. Becker and W. S. Cleveland, "Brushing scatterplots," *Technometrics*, vol. 29, no. 2, pp. 127–142, 1987.

[57] J. Venable, J. Pries-Heje, and R. Baskerville, "A comprehensive framework for evaluation in design science research," in *International Conference on Design Science Research in Information Systems*. Springer, 2012, pp. 423–438.

[58] A. Holzinger, "Usability engineering methods for software developers," *Communications of the ACM*, vol. 48, no. 1, pp. 71–74, 2005.

[59] E. Murphy-Hill, *Programmer friendly refactoring tools*. Portland State University, 2009.

[60] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[61] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools?: An empirical investigation of stack overflow," in *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*. ACM, 2013, pp. 33–36.