Cloud-based Collaborative Software Development: A Concept for Managing Transparency and Privacy based on Datasteads

Roy Oberhauser
Computer Science Dept.
Aalen University
Aalen, Germany
roy.oberhauser@htw-aalen.de

Abstract - **Cloud-centric collaboration in (global) software development continues to gain traction, resulting in new development paradigms such as Tools-as-a-Service (TaaS) and Cloud Development Environments based on Software-as-a-Service (SaaS). However, both within and between clouds, there are associated security and privacy issues to both individuals and organizations that can potentially hamper such well-intentioned collaboration. This paper describes an inter-cloud security and privacy concept for heterogeneous cloud developer collaboration environments that pragmatically addresses the distributed transmission, aggregation, storage, and access of events, data, and telemetry related to development projects, while giving individual developers fine-granularity control over the privacy of the data collected. To this end, the concept adapts an existing collaborative development and measurement infrastructure, the Context-aware Software Engineering Environment Event-driven framework (CoSEEEK) to support cloud-based event aggregation capabilities. The results and discussion show its practicality and technical feasibility while presenting performance tradeoffs for different cloud configurations. The concept enables infrastructural support for privacy, trust, and transparency within development teams, and could also support compliance with national privacy regulations in such dynamic and potentially global collaborative environments.**

Keywords - cloud-based software engineering environments, cloud-based software development collaboration, software project telemetry, privacy, security, trust.

I. INTRODUCTION

This article extends previous work in [1]. Global software development (GSD) [2] is increasingly taking advantage of cloud-based software applications and services [3] and realizing its collaboration potential. Data acquired and utilized during the software development and maintenance lifecycle is no longer necessarily locally controlled or even contained within an organization, but may be spread globally among various cloud providers with the acquired data retained indefinitely. Tools-as-a-Service (TaaS) [4] and cloud mashups will enable powerful new applications that utilize the acquired SE data [5]. And while the technical landscape is changing, the corporate landscape is also. A 2005 survey of American corporations conducted by the American Management Association showed that 76% monitored employee Internet connections, 50% stored and reviewed employee computer files, and 55% retained and reviewed email messages, with a rapidly increasing trend [6].

The ability to measure and minutely observe and track software developers during their work is becoming technically and economically viable to employers, managers, colleagues, virtual teams, and other entities. While metrics can be useful and provide a basis for improvements, be it at the organizational level (e.g., the CMMI Measurement and Analysis process area [7]), at the project level via automated software project telemetry (e.g., [8]), or for personal improvement (e.g., Personal Software Process [9], [10]). Unintended effects and abuse are also possible, such as [11] and [12], misuse of publicized information, misuse by competitors, mobbing, etc. While software services and apps developed by vendors for public customers typically attend to user privacy due to their longevity, mass accessibility, and regulatory and legal scrutiny, relatively little attention has been paid to the privacy needs of software developers, an estimated 17 million worldwide [13].

Consequently, privacy is becoming a looming concern for software developers that faces unique technical challenges that affect their collaboration. These challenges include: a highly dynamic technical environment typically at the forefront of software technology and paradigms (e.g., new languages, compilers, or platforms); diverse tools (for instance, [4] alone identifies 384); heterogeneous project-specific tool chains (e.g., application lifecycle management, version control systems, build tools, integrated development environments, etc.). Additionally, because development environments are often project-centric (unique and perhaps short-lived undertakings), the extra hassle and overhead for addressing developer privacy may seem to be an unnecessary hindrance to project progress and thus not be addressed at the management level. When multinational coordination (e.g., offshoring) is involved, multiple regulatory issues may apply and add to the complexity, etc. Developers may thus have little leverage and currently few technical options or suggestions for having their concerns addressed. Any privacy options should thus be economical and practically feasible, yet due to the dynamic technological nature of collaborative development environments (CDEs), standardization is unlikely or will be highly challenging.

To enable collaboration, the trust climate plays a vital role in the success of virtual and distributed teams [14], and trust and transparency are considered vital values for effective teams and collaboration [15][16]. Where trust exists (consider Theory Y [17]), collected data can be utilized collaboratively to enhance team performance [18], for instance by utilizing event data to coordinate and trigger actions and to provide insights, whereas where data is misused as an instrument of power, monitoring, or controlling (consider Theory X [17]), individuals require mechanisms for protection. Since the technical development infrastructure cannot know a priori what trust situation exists between some spectrum of complete trust to complete distrust, infrastructural mechanisms should support collaboration within some spectrum, while allowing the individuals and organizations to adapt their level of data transparency to the changing trust situation. Not all actors involved may have an issue with metric collection, while those who favor complete transparency may presume that those voicing privacy issues may seem to be "hiding something".

Privacy is control over the extent, timing, and circumstances of sharing oneself. Cloud service users currently have few personal infrastructural mechanisms for retaining and controlling their own personal data. Diverse privacy regulations are applicable within various geographic realms of authority. Various (overlapping) (multi-)national laws and regulations may apply to such (global) collaborative cloud contexts. For instance, Germany has a Federal Data Protection Act, the European Union has a Data Protection Directive 95/46/EC, and within the United States, various states each have their own internet privacy laws. Many privacy and security principles are typically involved, including notice, consent, disclosure, security, earmarking, data avoidance, data economy, etc. Various challenges for security and privacy in cloud environments remain [19][20]. In the interim, pragmatic infrastructural approaches are needed to deal with the issues in some way.

As in the initial paper [1] on which this extended version is based, the contribution includes elucidating the requirements and describing a solution concept for pragmatically addressing various privacy and security concerns in cloud-based dynamic heterogeneous CDEs. The solution is based on service layering, introduces distributed *cloud-based datasteading* for individuals, and mediates trust via brokers. Its technical feasibility and performance tradeoffs are investigated in an initial case study. Additional contributions of this extended version include a discussion, details, and evaluation of an aggregator implementation supporting push-based event collection from personal datasteads.

This paper is organized as follows: the next section describes the assumptions and requirements for a solution, while Section III describes related work. In Section IV, the solution concept is introduced, with the following section providing details of a technical implementation based on the concept. Evaluation results are presented in Section VI. Section VII provides a discussion, which is then followed by a conclusion and description of future work.

II. REQUIREMENTS

The following requirements, assumptions, or constraints (denoted by the prefix *R: in italics*) were elicited from the primary problems, goals, and challenges introduced in the preceding section, and are considered to be generally applicable for any conceptual solution. They are summarized here to highlight key considerations in the solution concept.

Multi-cloud configurability (*R:MultCld*): in view of GSD, inter-organizational collaboration, and the long-term nature and scale of certain development projects, any solution should support private clouds (*R:PrivCld*), public clouds (*R:PubCld*), and community clouds (*R:CmtyCld*) for a wide array of deployment options.

Cloud portability or provider-specific cloud API independence (*R:CldPort*) should be supported to avoid cloud provider lock-in and allow wider adoption and applicability. Development teams tend to want choices in their tooling and infrastructure to optimize and tailor their project or situation based on costs or risks (business, quality-of-service, potential espionage risks, etc.). If this is challenging because cloud vendors do not want to change or agree to some common interoperability standard, adaptation techniques such as bridging, brokers, or mediators could be used to support common infrastructure functionality.

Cloud compatibility (*R:CldCmpat*) with current public cloud provider and private cloud APIs and services should be supported. This entails avoiding exotic requirements for special configurations that would constrain its practical usage, such as refraining from special hardware requirements such as the Trusted Platform Module (TPM), or obtuse software languages, platforms, operating systems, or communication mechanisms that, while perhaps increasing privacy or security to some degree, might nevertheless hinder overall adoption of such an approach because such configurations require too much effort or become too unwieldy or difficult to implement and maintain.

Single tenancy (*R:1Tenant*) in the personal (developer's) cloud should be supported to reduce risk (e.g., to avoid a misconfiguration compromising a much larger set of tenants simultaneously) and to avoid access by organizational administrators, which can involve an additional trust issue beyond the project level.

Disclosure (*R:Dsclsr*): three fundamental levels of disclosure shall be supported: non-disclosure, anonymized disclosure, and personally-identifiable disclosure to specific aggregators. This allows the developer to adapt the disclosure of events and data to the trust situation of a specific project or group.

Sensor Privacy (*R:SnsPriv*): It is assumed that any client-side and server-side sensors, (e.g., version control system sensors) distribute personally-identifiable events according to a privacy concept, or are at least configured in such a way that they only transmit their events securely directly to a single datastead.

Entity-level privacy control (*R:EntityCtrl*): the granularity of privacy is controllable by the entity involved or affected, be it persons, teams, organizations, projects, etc., and flows from bottom-up (from persons to teams) and across for similar levels (e.g., between teams or between organizations). Top-down controls can only restrict privacy, e.g., in the case where organizations no longer trust each other (perhaps due to legal action), they cannot forcibly increase the disclosure levels of lower entities.

Restrict network access (*R:R:PrivNtwk*) to collaboration participants only, e.g., via Virtual Private Networks (VPN), to reduce the accessibility of the communications to the collaborators only. It may also be useful even within a larger organization's intranet to reduce accidental leakage risk.

Secure communication (*R:SecComm*) can be used to protect internal or external data transmission. This may be considered useful even within a VPN for retaining personal privacy.

Basic security mechanisms (*R:BscSec*): this specifies the reliance on widely-available off-the-shelf security mechanisms (e.g., HTTPS), without any dependence on specialized or exotic hardware or software security platforms (e.g., Trusted Platform Module) or research-stage mechanisms that would constrain its practicality.

Encryption (*R:Encrypt*) can be used to protect data accessibility and storage.

Trusted code implementation (*R:TrstCd*): Open source and/or independent code audits together with secure distribution mechanisms (e.g., via digital signatures from a trusted website) provide assurance that the code implementation can be trusted.

Remote runtime *code integrity verification* (*R:Intgrty*) should be supported to allow agents (e.g., automated temporally random auditing requests or manually initiated user requests) to detect any tampering with the implementation, sensors, configuration, or the compromise of any privacy safeguards.

It is generally assumed that the environment and culture within an organization and between organizations is fundamentally one of mutual respect, benefit, and trust, with appropriate IT policies that reflect this, and that explicit surveillance and undermining tactics, tools, etc. are not tolerated or utilized to undermine employee personal privacy. In other words, this solution is not meant to address privacy and security at a hacker, professional, corporate, or espionage level, but rather to give developers choices in sharing their personal event and metric data with others in differing personal and project trust contexts and where they understand and know how that their collected data will be used to enhance productivity collaboratively. It has been said "you get what you measure," and, when applied to individuals, the repercussions could be greater the more exposure certain personal data has. This could result in (un)intentional manipulation or misinterpretation of the data out of context, in one direction to perhaps show off, and could negatively affect other hitherto positive interactions. E.g., if one measures individual programmer productivity and broadcasts this, then such desired behaviors as helping others or team or quality issues may be diminished or ignored. Other team members may very well be crucially supporting the development effort but not in ways currently being measured.

In summary, a primary tenet here is that organizations and teams want to support privacy freedom for individuals, that they support and value self-organizing teams, and that they do not wish to hinder electronic collaboration and communication. While together the aforementioned elucidated requirements are not intended to be sufficient or complete, they nevertheless provide a practical basis for considering and comparing solution concepts and can be useful for furthering discussion.

III. State of the Art

In the area of global software development, [4] discusses support for TaaS and [21] Software-as-a-service in collaborative situations. Neither go into detail on various privacy issues, nor is support for various aforementioned requirements, e.g., for individuals (*R:EntityCtrl*). Example industrial offers for cloud-based collaboration include Atlassian OnDemand and CollabNet CloudForge. Individual privacy control (*R:EntityCtrl*, *R:Dsclsr*) do not appear to be supported.

Work on more general multicloud collaboration includes [5], which similarly supports opportunistic collaboration without relying on cloud standardization based on the use of proxies. However, aspects such as (*R:BscSec*, *R:Intgrty, R:EntityCtrl*) were not considered and a technical implementation was not investigated.

Work in the area of standardization and reference architecture includes [22], which mentions privacy but fails to prescribe a solution. [23] lists various security and interoperability standards and their status, but their maturity and market penetration when considering (*R:MultCld*) and (*R:CldCmpat*) remain issues.

Various general cloud security mechanisms have been proposed. Privacy as a Service (PasS) [24] relies on secure cryptographic coprocessors to provide a trusted and isolated execution and data storage environment in the computing cloud. However, its dependency on hardware within cloud provider infrastructure hampers (*R:CldCmpat, R:CldPort,* and *R:BscSec*). Data protection as a service (DPaaS) [25] is intended to be a suite of security primitives that enforce data security and privacy and are offered by a cloud platform. Yet this would inhibit (*R:CldPort*). Other work such as [26] describe privacy-preserving fine-grained access control and key distribution mechanisms, but are not readily available for a pragmatic approach that is usable today (*R:BscSec*).

IV. Solution Concept

For a cloud-based context-aware collaboration system to have satisfactory utility, it will depend on some type of event and data collection and communication facilities. Thus, this foundational infrastructure should be equipped with basic trust and security mechanisms such that upper-level services like context-awareness and collaboration can ensue without impinging on privacy.

Thus, to provide a flexible solution for achieving privacy control in such environments, a primary principle in the solution concept is the application of the *Service Layer* design pattern [27] to provide a decoupling and separation of concerns as shown in Figure 1. The lower conceptual Event and Data Services Layer includes event and/or data services for an entity (person/team/organization), including acquisition, storage, retention, and dissemination, while the upper Collaboration and Tools Services Layer includes CDE and tool services. The upper layer services utilize lower layer data to provide collaboration, data sharing, analytics, telemetry, contextual guidance, and other value-added services. Any single entity would have more limited privacy control mechanisms.

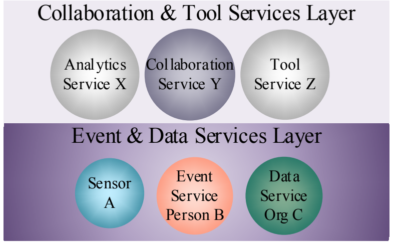

Figure 1. Services Layer Pattern.

A second solution principle is the introduction of a *datastead*, shown in Figure 2. Loosely analogous to the concept of homesteading or seasteading, it provides an entity with both a certain degree of data isolation and control for some area. In this case, some entity (be it an individual or some unit) manages and controls clearly delineated data resources in the cloud for which they have or receive responsibility and ownership rights. The technical implementation of a datastead can be in the form of a personal cloud in the case of an individual, or an area within a private cloud for an organization. It is thus clear to the individual or entity that they have complete control over personal (or entity) event and data storage that is kept separate under their personal (or entity) jurisdiction. Each datastead can pass data to one or more other datasteads (such as one belonging to a team) or directly to (usually one) community cloud where it can be processed and utilized to enhance collaboration. A configuration with successive, staged, or pipelined datasteads, while not required, can support the need for entity level privacy and disclosure control from the lowest levels to the highest levels in organizations (bottom-up). Community clouds may also successively pass data on to larger community clouds if desirable to the providing community. For instance, academic research communities could access and analyze this data for multiple projects.

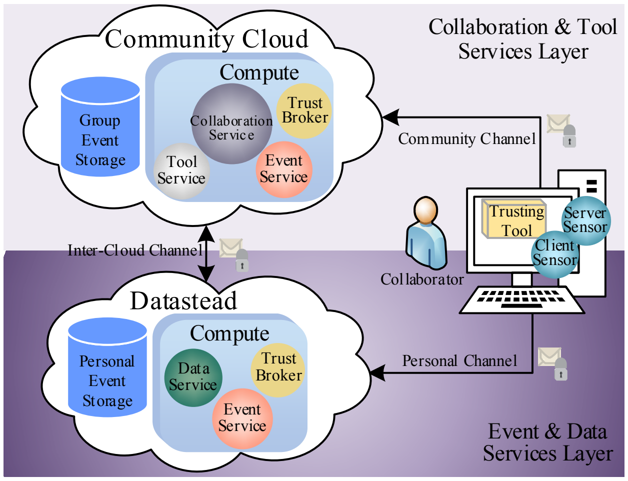

Figure 2. Generic Solution Concept.

The third principle is the inclusion of a Trust Broker that mediates between service and data access, acting as both a cloud service broker (for interoperability with various tools) and cloud security broker (for security) between layers. Akin to the Trusted Proxy pattern [28] and Policy Enforcement Point [29], it constrains access to protected resources and allows custom, finely-tuned policies to be enforced (*R:EntityCtrl*). Rules can be used to configure and distinguish/filter access by event types, timeframes, projects, etc. It provides secure communication mechanisms (*R:SecComm*) to authenticate and authorize data acquisition and data dissemination in the datastead, as well as interoperability mechanisms for various collaboration and tool services. Only client requests from preconfigured known addresses are accepted. A management interface to the Trust Broker provides the datastead owner with policy management capabilities. It also supports data anonymization on a per request basis if so configured. For secure storage, the Trust Broker encrypts (*R:Encrypt*) acquired events and data (Encrypted Storage pattern [28]) to prevent unauthorized access by administrators or intruders, and protects access to the encrypted storage typically on a single port (Single Access Point pattern [29]). The Trusting Broker supports runtime code integrity (*R:Intgrty*) via remote attestation, and a client, called the Trusting Tool, can be invoked periodically or based on certain events to ensure that the Trust Broker has not been tampered with.

As to transmission, a Personal Channel transmits events from sensors to the personal datastead. The Inter-Cloud Channel transmits personal or anonymized events to one or more Community Clouds. The Community Channel is optional and can be used, e.g., for impersonal sensors (e.g., team build server) or perhaps in special situations when duplication and parallel transmission of personal events for reliability or performance is desired and approved. Secure Channels and Secure Sessions [29] are used to protect the transmission between the sensors and the datastead (the Personal Channel), between sensors and the Community Cloud (Community Channel), as well as between the datastead and any collaboration and tool services (Inter-cloud Channel). For a community cloud, a VPN is used to limit network access to collaborators in the community only.

V. TECHNICAL IMPLEMENTATION

To determine the technical feasibility of the solution concept and provide a concrete case study, the solution concept was applied to an existing heterogeneous CDE called the Context-aware Software Engineering Environment Event-driven framework (CoSEEEK) [30], which had hitherto not incorporated privacy or security techniques. CoSEEEK's architecture and integrated technologies are shown in Figure 3. Its suitability is based on its portability (use of mainly Java and web-based languages), use of non-commercial technologies described below, its reliance on common distributed communication mechanisms such as RESTful web services, and its heterogeneous tool support. Additional technical details on CoSEEEK can be found in [31][32].

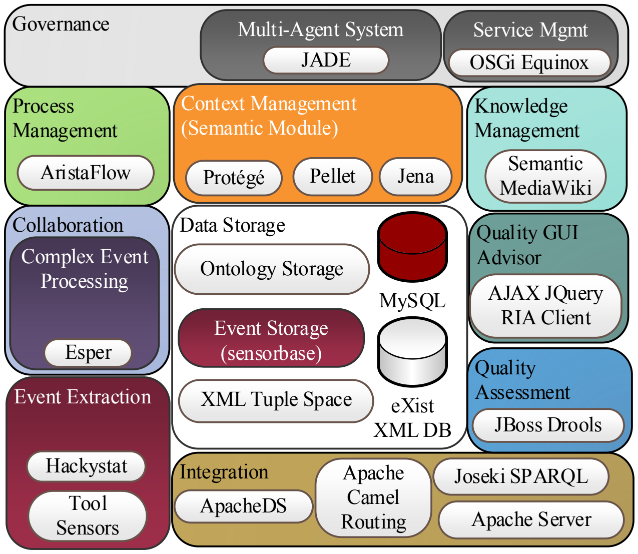

Figure 3. CoSEEEK Architecture (affected areas shown in red).

For event acquisition, CoSEEEK relies on the Hackystat framework [33] and its SE tool-based sensors (e.g., Ant, Eclipse, Visual Studio) for event extraction and event storage (shown in red in Figure 3). Hackystat does not currently provide extensive security and privacy mechanisms. For an insight, [34] briefly describes some of its security issues.

Service Layer Separation: the Hackystat-related elements (shown in red) were hereby separated into the Event and Data Services Layer and the remaining elements were placed in the Collaboration and Tools Services Layer.

Cloud configuration: To meet (*R:MultCld*, *R:CldCmpat*, *R:CldPort*), two different cloud platforms were utilized in isolation. To represent a public IaaS cloud provider configuration (*R:PubCld*), Amazon Web Services (AWS) was used, using Elastic Compute Cloud (EC2) for computing services, the Elastic Block Store (EBS) for storing configuration files and XML database, and the Relational Database Service (RDS) that holds the sensorbase.

To represent a private cloud (*R:PrivCld*) or community cloud (*R:CmtyCld*) deployment, OpenStack was used with Compute used for computing and Object Storage used in place of EBS storage. Since nothing directly equivalent to AWS RDS was available, we configured a Compute instance with Object Storage that contains a MySQL Server database and for single tenancy (*R:1Tenant*) one Compute instance per developer with access restricted to the developer.

Trust Broker: the Trust Broker supports (*R:Dsclsr*) was implemented in Java using a REST framework. An example of a query that can be sent is the following, specifying the project via the sensorbase_id, the timeframe, the sensor data type, the tool, and its uri source.

```
   GET
/trustbroker/sensordata/{sensorbase_id}?
startTime={startTime}&endTime={endTime}&
sdt_name={sdt_name}&tool={tool}
   &uriPatterns={uriPatterns}
```

Encryption of events (*R:Encrypt*) can be optionally configured. For encryption of arriving events and decryption of events on authenticated and authorized retrieval, Java's AES 128 and the SHA-256 hash algorithm were used (*R:BscSec*). One reason for encrypting the storage is that it provides an additional form of protection, should, e.g., a provider's agent or intruder gain access.

The measurement database, called sensorbase in Hackystat, required a few minor adaptations. For (*R:Dsclsr*) to support anonymization, the HACKYUSER table was extended to include an anonymization flag that is checked before responding, replacing a userid with anonymous. In order to support HTTPS connections, the sensorbase client (*R:SnsPriv*) was modified and rebuilt, requiring any sensors to utilize this modified jar file. HTTPS (*R:BscSec*) was used to secure all three communication channels (personal, community, and inter-cloud) (*R:SecComm*). Additional properties were added to indicate the location of the keystore. SSH was used to configure and manage each cloud. Security groups were used in both AWS and OpenStack.

A. Aggregator

To improve cloud-based performance, we adapted the solution concept from our initial paper to remove the ongoing querying of the datastead by the Trust Broker in the community cloud. Instead, a client-based push approach for event transmission was implemented. A blacklist within the client Trust Broker filters or anonymizes the types of events that are passed on and made available. The aggregation of the events now avoids polling the clients. For this, the client Trust Broker is responsible for tracking which events were already successfully transferred and which still need to be sent to any Aggregator in the Trust Broker on a Community Cloud.

The interaction between components for the transfer of events is shown in Figure 4. Sensors send their events on a push basis to the Hackystat sensorbase located in the datastead. The datastead Trust Broker periodically queries the sensorbase for events newer than its last transmission to any particular Aggregator. Once an event is fetched, it is checked against a blacklist to determine if it should be blocked or anonymized. Then the datastead Trust Broker pushes the event via its REST connection to the Aggregator within the Trust Broker residing on the Community Cloud, where it is persisted and can be processed by various upper-level services in CoSEEEK. This is repeated in a loop until the latest event has been sent.

There are valid arguments for maintaining either a whitelist or blacklist, depending on what standpoint one takes (send most data vs. send almost no data), the extent of the associated rules, as well as the consideration of what should happen if something is not specified. In the case of a blacklist where something is missing or was misconfigured, that would imply that events would slip through.

While a whitelist could have been used, for simplification of the implementation for demonstration purposes we chose to use a blacklist, since we presume that developers will more likely know exactly what sensors they want to block or anonymize (i.e., blacklist) from the community more than they will care to know and manage all the diverse and possible sensors (whitelist).

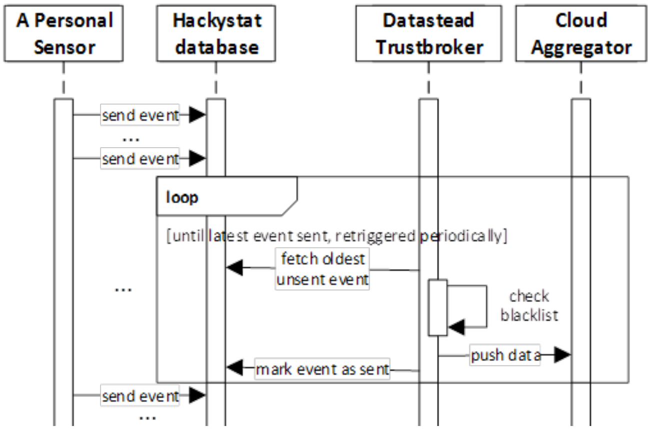

Figure 4. Sequence diagram showing push-based aggregation.

The file blacklist.xml specifies which events should be blocked or anonymized, whereby blocking is the default. The boolean tag `<Anonymize>` controls anonymization, but can be explicitly set to `false` as a kind of "unblock" to allow a specific event to be unblocked when most others are blocked.

If no events should be anonymized or blocked, then the list is empty as shown below.

```
<Blacklist>
</Blacklist>
```

By default, if a tool is listed all events from that tool sensor are blocked and remain in the datastead. For example, to block all events from the eclipse tool sensor, it would be specified as follows:

```
<Blacklist>
  <Tool>
    <Name>Eclipse</Name>
  </Tool>
</Blacklist>
```

To anonymize, for example, all events from the eclipse tool sensor, it would be specified as follows:

```
<Blacklist>
  <Tool>
    <Name>Eclipse</Name>
    <Anonymize>true</Anonymize>
  </Tool>
</Blacklist>
```

For example, to by default block all the other Eclipse tool events, but anonymize only the Eclipse File Open events, it would be specified as follows:

```
<Blacklist>
  <Tool>
    <Name>Eclipse</Name>
    <Property>
      <Anonymize>true</Anonymize>
      <Key>Subtype</Key>
      <Value>Open</Value>
    </Property>
  </Tool>
</Blacklist>
```

To anonymize all Eclipse events, and by default block the Eclipse File Open events, one specifies additional properties, as shown here:

```
<Blacklist>
  <Tool>
    <Name>Eclipse</Name>
    <Anonymize>true</Anonymize>
    <Property>
      <Key>Subtype</Key>
      <Value>Open</Value>
    </Property>
  </Tool>
</Blacklist>
```

To anonymize all Eclipse events except the File Open events, for example, the following would be specified:

```
<Blacklist>
  <Tool>
    <Name>Eclipse</Name>
    <Anonymize>true</Anonymize>
    <Property>
      <Anonymize>false</Anonymize>
      <Key>Subtype</Key>
      <Value>Open</Value>
    </Property>
  </Tool>
</Blacklist>
```

To manage what event to push, a simple event timestamp reference that is persisted tracks the last event successfully retrieved from the Hackystat sensorbase and that was either blocked due to the blacklist or transmitted to a specific Aggregator on a per event basis. Should an error during transmission occur, the client is responsible for retransmitting. Should the Aggregator become unavailable, the client will continue to retry rebuilding a connection until it succeeds in pushing the events not yet successfully transmitted in the order of their occurrence (timestamp). No separate queue is maintained and all events are stored in the sensorbase.

B. *Remote Attestation*

To implement remote attestation, on the client-side, a user configures the Trusting Tool with the expected checksum value (provided, e.g., by the admin or a trusted website), version, and the interval for rechecking. On the service side, a REST interface sensorbase/checksum was added that loads the local adapted sensorbase.jar file, computes the SHA-256 hash value using `java.security.MessageDigest`, and returns this value and the sensorbase version to the Trusting Tool. While not foolproof, since any unauthorized access on the server or client could allow spoofing, it provides an additional level of confidence. Various stronger jar file tampering technologies could be employed if needed, such as componio JarCryp bytecode encryption.

VI. Evaluation

The case study evaluated the technical feasibility of the concept based on the technical implementation. However, security and privacy are highly contextually dependent on the expectations, requirements, environment, risks, policies, training, available attack mechanisms, implementation details (bugs), configuration settings, etc. Therefore, making a comprehensive formal assessment in this area is difficult. So the assumption is made that the prescribed privacy and security mechanisms suffice or are balanced for current developer needs in developer settings.

Since CoSEEEK is a reactive system, the ability to respond adequately to contextual changes via events is dependent primarily on event latency. Cloud networking, additional network security mechanisms, and the additional delay incurred by inserting datastead nodes could negatively affect responsiveness, and thus this infrastructural level of event latency was the primary focus of the evaluation.

To elaborate, as CoSEEEK is a process-centered software engineering environment, any events that arrive too late to be contextually relevant can cause CoSEEEK-triggered actions or responses to be irrelevant and thus ignored. Developers also tend to be impatient when a guidance system is not providing relevant and applicable guidance for the context when expected, and they will continue on without it and perhaps begin to ignore it. As to event volume, events generated by any single developer's actions are typically sporadic and not highly voluminous. If a sensor is overly vociferous in relation to the amount of developer activity, it can typically be configured to eliminate redundant events or to summarize events. If this capability is not built in, a complex event processor (e.g., Esper) can be utilized to reduce the load on the network and aggregator in larger project environments.

A subjective evaluation by developers in an industrial setting was considered but not feasible at this time due to resource and schedule constraints, and is included in future work.

A. Security Overheads

To determine security overheads, the Client PC (for use by a developer) has an i5-2410M (2.3-2.9 GHz) dual core CPU and 6GB RAM with 32-bit Windows XP SP3. The network consists of gigabit Ethernet and two 1 Gbit connections from the university campus in Germany to the Internet Provider.

Representative for a private (*R:PrivCld*) or community cloud where a datastead could also be placed, the OpenStack configuration (OSCfg) consisted of a local intranet server with an i5-650 (3.2-3.4GHz) dual core CPU, 8GB RAM, and 64-bit Ubuntu Server 12.04. The OpenStack Cloud Essex Release was installed on the Server via DevStack and the Compute instances also ran Ubuntu Server 12.04. MySQL v. 5.5.24 was used for Hackystat sensorbase storage in a Compute instance.

As a public cloud provider (*R:PubCld*) representative, a free AWS configuration (AWSCfg) was chosen. It consisted of t1.micro EC2 instance types located in US-EAST-1d (Virginia) with 613 MiB memory, up to 2 EC2 units (for short periodic bursts) with low I/O performance running 64-bit Ubuntu Server 12.04. MySQL v. 5.5.27 was used for the Hackystat sensorbase storage in AWS RDS.

Common software included Hackystat 8.4 with the Noelios Restlet Engine 1.1.5 and JDK 1.6.

Typical network usage scenarios were considered, thus no optimizations were applied to any configurations nor was an artificially quiet network state created. All results are the average of 10 repeated measurements (with one exception noted below). A secure configuration denotes using the TrustBroker via HTTPS (*R:SecComm*) with encrypted storage (*R:Encrypt*), and an insecure configuration means HTTP without a TrustBroker. VPN (*R:R:PrivNtwk*) overheads were not measured.

To determine delays from the client to the datastead in cloud variants, on the client PC the Ant build tool was invoked, causing the Hackystat Ant sensor to send one XML event to the Server (a write in the remote sensorbase) consisting of 235 bytes of event data plus 73 bytes of network protocol overhead. The measured latency values are shown in Table I and Figure 5.

TABLE I. LATENCY (IN MS) FOR SENDING AN EVENT (235 BYTES) FROM THE CLIENT PC TO THE SERVER SENSORBASE

Insecure Private Cloud (ms)	Secure Private Cloud (ms)	Secure AWS Cloud (ms)
214	389	608

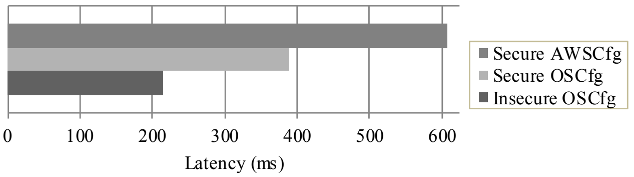

Figure 5. Latency (in ms) for sending an event (235 bytes) from the client PC to the server sensorbase.

Once events are in the datastead, then latencies incurred between cloud computing instances are of interest, since collaboration services or tool services will require this data. The measured values are shown in Table II and grouped by security mechanisms in Figure 6.

A grouping by cloud type is shown in Figure 7. For AWSCfg, a single query for 67 events (15818 bytes) between two EC2 instances took 78 ms on average via HTTP and 84 ms over HTTPS. In a secure configuration the retrieval took 347 ms. For OSCfg between two Compute instances, a single query took 38 ms to return 22 events (5243 bytes). Note that HTTP insecure reads in the private cloud had two anomaly values (178 and 210 ms) that would have changed the average from 38 to 69, and were also far larger than any secure value measurements. Thus, these two measurement values were removed, and the average created from the remaining 8 values. These large latencies could perhaps be attributed to a network, disk, operating system, or OpenStack related issue. Continuing with the measurements with 39 events (9238 bytes), HTTPS requests took 60 ms while in the secure configuration it averaged 61ms. The overhead of the privacy approach is the addition of SSL, brokering a second SSL connection, and encryption. For the OSCfg, the difference of TrustBroker and decryption showed on average only a 1ms difference to that with purely SSL. One explanation could be that the extra overhead is minimal compared to the data transfer delays between OpenStack instances, but further investigation of OpenStack internals and performance profiling would be needed to clarify this.

TABLE II. PRIVATE VS. PUBLIC CLOUD INTER-COMPUTING INSTANCE QUERY LATENCIES (IN MS)

	Inter-OSCfg Latency (ms)	Inter-AWSCfg Latency (ms)
Insecure	38	78
HTTPS	60	84
Secure	61	347

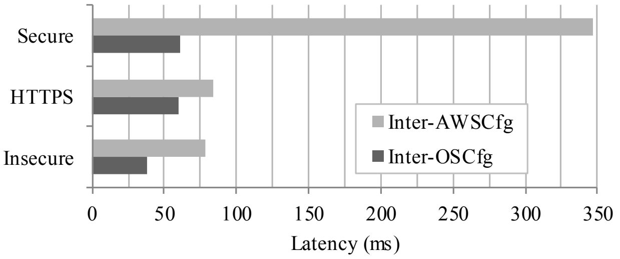

Figure 6. Private vs. public cloud inter-computing instance query latencies grouped by security (in ms).

Based on the results shown in the above figures, the use of the secure configuration of the OSCfg within a private or even a community cloud setting would appear to have acceptable performance overhead for cloud-centric collaborative development work, and distributed retrieval from datasteads is viable for responding to changes in the collaborative situation. On the other hand, the use of the secure configuration in the public cloud (AWSCfg), as shown in this perhaps worst case as a free offshore minimal public cloud setting, incurs substantially higher network latencies. Obviously, choosing geographically close locations when possible is recommended. Also, provisioning sufficient computing and I/O resources support to deal with the additional inter-cloud and security mechanism overheads would also reduce such lags in public cloud configurations. Optimizing this area could yield performance improvements but may incur additional financial costs.

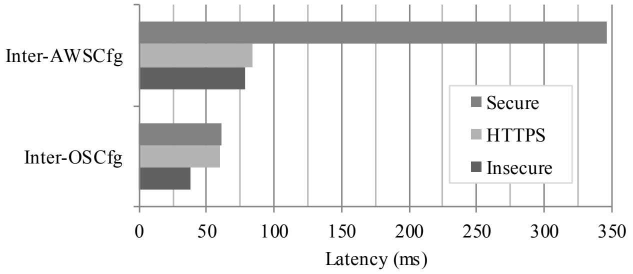

Figure 7. Inter-cloud query latency grouped by cloud type for different degrees of security (in ms).

To determine the remote attestation overhead, the Trusting Tool was measured on the PC using the AWSCfg over SSL. The average request-response latency was 702 ms. On the server, this involved loading and calculating the SHA-256 hash value for the 5.5 MB large sensorbase.jar file. Thus, the attestation mechanism of the remote cloud instance could be configured to be automatically invoked periodically by client-side sensors at regular intervals in a separate thread or process so as not to interfere with other network communication.

B. *Aggregator Performance*

In order to remove the query of datasteads by the Trust Broker Aggregator, the implementation was changed to a client-based push approach as mentioned in the previous section. The aggregator push implementation was measured separately to determine its performance and adequacy. For this a client PC served as the datastead. The following hardware setup was used for these measurements: the client was a Lenovo ThinkPad X201T with 2GHz Intel Core i7 L620 and 4GB RAM. The local server consisted of a PC with a 3.30GHz Intel Core i3-3220 CPU with 4GB RAM. Amazon AWS T2.micro consisted of 1 VCPU with 1GB RAM. All used a 64-Bit Ubuntu version 14.04. The network connection consisted of a 1 Gbit LAN between the PC and server locally and 2.5Mbit upstream and 50Mbit downstream to the internet provider. HTTP with REST was used for these measurements using Jersey 2.10 and the Java Runtime Environment 7.

Since the filtering of events will not consume significant wall clock time in comparison to the network aggregation, event filtering and anonymization were disabled. 128 events were pushed to the Aggregator from the client. Since events are not likely to be excessively large, events of 256 and 512 bytes in total length were used for comparison. The results are shown in Table III and Figure 8.

No significant latency differences due to a larger event size were detected on the local network. This can be explained in that the primary overheads involved are not related to content analysis or processing of data within the packets or events since this was not performed, and that the high network transmission rate available made the additional payload insignificant.

The latency durations indicates that potential exists here for performance optimization, but due to time and resource constraints a more thorough analysis of these initial results using CPU profiling and network sniffers could not be performed.

TABLE III. LOCAL VS. PUBLIC CLOUD NETWORK AGGREGATION LATENCIES (IN MS) FOR 128 EVENTS

Event size (bytes)	Local Aggregator Duration (ms)	AWS Aggregator Duration (ms)
256	7528	10448
512	7527	10617

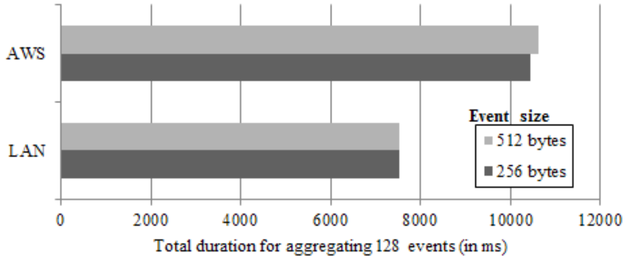

Figure 8. Aggregation network latencies (in ms) in local vs. public cloud settings for 128 events.

In summary, the evaluation showed that network latencies incurred by the solution concept are most likely insignificant for collaboration in PrC settings, but that security overheads in global PuC settings may require optimization attention to minimize their effects.

VII. DISCUSSION

Telemetry and metrics play a vital role in providing a data basis for assessing areas for improvement, for benchmarking against other organizations or projects, as a basis for root cause analysis, and for determining the effects of any improvement initiatives.

When the trust environment in an organization or project is healthy, then the sharing of event data and associated metrics can be used to support tighter collaboration, streamline interactions, and be used in retrospectives for analysis to support and verify improvements and best practices. However, when the trust environment is degraded or non-existent, then a forced sharing of detailed personal-level data may result in additional inefficiencies due to psychological or motivational effects, circumvention or abuse of such a system, or adapting behaviors or sensors to intentionally providing misleading data to make certain people look good and/or others look worse.

The Tuckman model [35] for the development of small groups may be useful to illustrate the difficulties as teams transition in their interactions and the associated change in the trust among members, from forming to storming, norming, performing, and then adjourning. The concept in this paper can adjust for increasing trust, from blocked to anonymous to personalized events. Should the trust situation decline, it can support adjustments in the policies from that point on for new events. Any events already disclosed will however remain so unless the community cloud is manually cleansed by an administrator.

If we look beyond software developers and at the broader picture of employees in organizations that intentionally monitor their employees, they are then likely to utilize surveillance products that were intentionally built for this purpose and will likely not explicitly involve their employees. Addressing privacy in such situations is beyond the technical scope of the concept and approach in this paper and will likely need to rely on regulatory mechanisms to balance the rights and responsibilities of the parties involved.

However, due to the dynamics of projects and development environments, and the freedom and influence or empowerment developers often have, software developers are in a unique position to influence the use of measurements for team improvement while balancing the amount of transparency and personal measurement to the shifting trust environment. Personal empowerment over personal measurement data may allow developers to embrace the adoption and inclusion of personal metrics and enhance the productivity of teams without the negative impacts of forced submission to measurement collection. The adoption in open source projects, for example, would allow deeper analysis and understanding, even if the metrics and events were anonymous. In the larger scheme of things, this could provide the software engineering research community with valuable additional data for (meta-)analysis and improvements in the hitherto semi-inaccessible data collection area associated with software development processes.

The approach in this paper intended to provide personal control mechanisms to developers to deal with privacy and trust reservations of developers towards the integration of sensors in their environments needed by collaboration and telemetry systems similar to CoSEEEK.

VIII. CONCLUSION AND FUTURE WORK

To address security and privacy in collaborative cloud development, this paper presented a practical concept with entity-level control of non-, anonymized-, and personally-identifiable disclosure for multiple cloud configurations. It can further both collaboration and trust by giving individuals transparency and control and allowing them to adjust disclosure to the changing trust situation. The paper contributes a practical basis for illustrating issues, eliciting awareness, community discussion, and may increase self-regulation and infrastructural privacy offerings. Organizations adopting such a privacy infrastructure show that they value and trust their employees, enabling them to reap mutual trust rewards. Also, one could envision, for instance, that an audited "we don't spy here" seal might help attract and retain developers for certain organizations.

The evaluation showed its technical feasibility and practicality, requiring only minimal adaptation of the CoSEEEK CDE. The Trust Broker enables fine granularity access control to personal data. Performance was sufficient in private cloud configurations, while public cloud configurations using additional security and privacy mechanisms may require optimization to ensure fluid collaboration situational response. The push-based aggregation supports black-list filtering and anonymization on a fine-grained event basis and better supports aggregating events when many clients are involved. The current implementation relies on the clients to avoid retransmission of events. Should this not suffice in practice, the Aggregator could be adapted to ensure that duplicates are not stored, e.g., by using a hash value or checksum for each received event, and any new events compared with all previous event hash values, although this would add some additional overhead.

Limitations and risks include: extending privacy/trust support within and across collaboration layer tools, non-detection/discovery of (un)intentionally unspecified/hidden sensors, data manipulation risk by datastead owners themselves, and provider-side access or manipulation risk. In the case of trust issues with the service provider, building your own datastead cloud server site could be considered. A concept for reliably maintaining and updating the software across the various datasteads in a trustworthy and efficient manner, with or without manual intervention by the datastead owner, should also be considered. Perhaps the updates should require some certification and could then be performed automatically if desired by the owning entity. In the end, developers will likely prefer low hassle solutions that still provide adequate privacy transparency and controls.

Future work includes an industrial field study, the inclusion of various data provenance and data integrity mechanisms to mitigate manipulation risk, and the investigation of enhanced remote attestation mechanisms. In the face of shifting privacy norms, infrastructural support for data confidentiality is needed to limit disclosure of distribution data beyond its original intent, like lifetime constraints, transitivity bounds, and claims-based access [36]. One challenge here is to deal with annulment or revocation of data already shared in the past when the trust situation degrades. Since service privacy is also a broader issue, development and adoption of global industry service privacy standards combined with independent privacy audits involving all service layers would enhance the trust of cloud-based data acquisition and service offerings.

ACKNOWLEDGMENT

The author wishes to acknowledge Roman Pisarew and Jürgen Drotleff and for their assistance with the concept, implementation, and measurements.

REFERENCES

[1] R. Oberhauser, "Towards cloud-based collaborative software development: A developer-centric concept for managing privacy, security, and trust," Proceedings of the Eighth International Conference on Software Engineering Advances (ICSEA 2013), pp. 533-538.

[2] S. Hashmi et al., "Using the cloud to facilitate global software development challenges," in Proceedings of the Sixth IEEE International Conference on Global Software Engineering Workshop (ICGSEW), IEEE, 2011, pp. 70-77.

[3] R. Grossman, "The case for cloud computing," IT professional, 11(2), 2009, pp. 23-27.

[4] M. Chauhan and M. Babar, "Cloud infrastructure for providing tools as a service: Quality attributes and potential solutions," in Proceedings of the WICSA/ECSA 2012 Companion Volume, ACM, 2012, pp. 5-13.

[5] M. Singhal et al., "Collaboration in multicloud computing environments: Framework and security issues," Computer, 46(2), IEEE Computer Society, New York, 2013, pp. 76-84.

[6] G. Nord, T. McCubbins, and J. Nord, "E-monitoring in the workplace: Privacy, legislation, and surveillance software," Communications of the ACM, 49(8), 2006, pp. 72-77.

[7] C. Team, "CMMI for development, version 1.3, improving processes for developing better products and services," no. CMU/SEI-2010-TR-033, Software Engineering Institute, 2010.

[8] P. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," IEEE Software, 22(4), 2005, pp. 76-85.

[9] W. Humphrey, "The personal software process: Status and trends," IEEE Software, 17(6), 2000, pp. 71-75.

[10] P. Johnson et al., "Beyond the personal software process: Metrics collection and analysis for the differently disciplined," Proceedings of the 25th international Conference on Software Engineering, IEEE Computer Society, May 2003, pp. 641-646.

[11] R. Irving, C. Higgins, and F. Safayeni, "Computerized performance monitoring systems: Use and abuse," Communications of the ACM, 29(8), 1986, pp. 794-801.

[12] J. Stanton, "Reactions to employee performance monitoring: Framework, review, and research directions," Human Performance, 13(1), 2000, pp. 85-113.

[13] M. Parsons, "The challenge of multicore: A brief history of a brick wall," EPCC News, Issue 65, University of Edinburgh, 2009, p. 4.

[14] T. Brahm and F. Kunze, "The role of trust climate in virtual teams," Journal of Managerial Psychology, 27(6), 2012, pp. 595-614.

[15] A. Costa, R. Roe, and T. Taillieu, "Trust within teams: The relation with performance effectiveness," European journal of work and organizational psychology, 10(3), 2001, pp. 225-244.

[16] T. DeMarco and T. Lister, Peopleware. Dorset House, 1987.

[17] D. McGregor, The Human Side of Enterprise. McGrawHill, New York, 1960.

[18] B. Al-Ani and D. Redmiles, "Trust in distributed teams: Support through continuous coordination," IEEE Software, IEEE Computer Society, 26(6), 2009, pp. 35-40.

[19] P. Louridas, "Up in the air: Moving your applications to the cloud," IEEE Software, 27(4), IEEE Computer Society, New York, 2010, pp. 6-11.

[20] H. Takabi, J. Joshi, and G. Ahn, "Security and privacy challenges in cloud computing environments," IEEE Security & Privacy, IEEE Computer Society, 8(6), 2010, 24-31.

[21] R. Martignoni, "Global sourcing of software development-a review of tools and services," In Fourth IEEE International Conference on Global Software Engineering (ICGSE 2009), IEEE Computer Society, 2009, pp. 303-308.

[22] F. Liu et al., NIST Cloud Computing Reference Architecture. NIST Special Publication, 500, 292, 2011.

[23] M. Hogan, F. Liu, A. Sokol, and J. Tong, NIST Cloud Computing Standards Roadmap. NIST Special Publication, 35, 2011.

[24] W. Itani, A. Kayssi, and A. Chehab, "Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures," In Eighth IEEE International Conf. on Dependable, Autonomic and Secure Computing (DASC'09), IEEE Computer Society, 2009, pp. 711-716.

[25] D. Song, E. Shi, I. Fischer, and U. Shankar, "Cloud data protection for the masses, " Computer, 45(1), 2012, pp. 39-45.

[26] M. Nabeel and E. Bertino, "Privacy-preserving fine-grained access control in public clouds," Data Engineering, 21, 2012.

[27] T. Erl, SOA Design Patterns. Pearson Education PTR, 2008.

[28] D. Kienzle, M. Elder, D. Tyree, and J. Edwards-Hewitt, Security Patterns Repository Version 1.0. DARPA, 2002.

[29] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, Security Patterns: Integrating Security and Systems Engineering. Wiley, 2006.

[30] G. Grambow, R. Oberhauser, and M. Reichert, "Enabling automatic process-aware collaboration support in software engineering projects," in Software and Data Technologies, Springer, Berlin Heidelberg, 2013, pp. 73-88.

[31] R. Oberhauser, "Leveraging semantic web computing for context-aware software engineering environments," Semantic Web, Gang Wu (ed.), In-Tech, Austria, 2010.

[32] G. Grambow, R. Oberhauser, and M. Reichert, "Contextually injecting quality measures into software engineering processes," the International Journal On Advances in Software, ISSN 1942-2628, vol. 4, no. 1 & 2, 2011, pp. 76-99.

[33] P. Johnson, "Requirement and design trade-offs in Hackystat: An in-process software engineering measurement and analysis system," Proc. First Intl. Symposium on Empirical Software Engineering and Measurement, 2007, pp. 81-90.

[34] P. Johnson, C. Moore, J. Miglani, and S. Zhen, Hackystat Design Notes. 2001.

[35] B. Tuckman, "Developmental sequence in small groups," Psychological bulletin, 63(6), 1965, p. 384.

[36] D. Reed, D. Gannon, and J. Larus, "Imagining the future: Thoughts on computing," Computer, 45(1), 2012, pp. 25-30.