VISDOM:
Data fetchers state of practice

Qentinel Quality Intelligence Oy
Henri Terho

Table of contents

Change History

Date	*Change*	*Name*	*Version*
22.12.2019	*Initial layout of the document and background*	*Henri Terho*	*0.1*
30.12.2019	*Data architecture*	*Henri Terho*	*0.2*
31.12.2019	*Date Fetchers*	*Henri Terho*	*1.0*

Background

At Qentinel we have been at the forefront of software quality assurance and the processes associated with it. We believe your business is only as good as the software driving it. Our customers have millions of critical devices that will be integrated into a huge number of systems in the future. To accomplish this requires a fierce amount of automation, new ways and insights. Qentinel ensures that your software is running flawlessly and creating value to your business. We accelerate your software development and improve the user experience with robotic SW testing. That is our specialty and we are the best at it.

We have great pride in our automated testing solutions that enable customers to deliver quality software even in the most critical of environments such as healthcare. Our software solutions have not been limited to just automated testing, but cover also project and business case management and customer and user experience. Our overarching goal has been to create a solution to control the total quality of the software project from the first software requirement to the day the solutions is removed from production usage. For us quality is much more than technical compliance.

From this standpoint, we have identified that to accurately measure software quality you have to import data from multiple sources and form a holistic picture of the whole software project. Collecting data from multiple different sources is a problem in itself, as there are multiple different types of data sources and data types. The frequency of the datapoints varies wildly and there are multiple different systems, which represent the same data object in a slightly different way. This results in problems to be solved in both data access and data homogenization.

Data homogenization is also an important part of making all this heterogenous data from multiple sources easy to access and process into information. As if we do not have an easy way to access all the collected data, we have not achieved much in fetching the data.

This document is organized into three parts. At first we speak about the data and its format. In the second part we talk about the different ways to fetch the data and the environments we have to interface with to access the data. In the third part we talk about the management of these different fetchers to organize them into a coherent data collection pipeline.

Data

Collecting data about a software project involves interfacing with multiple tools that are used in different phases of the software development pipeline. Examples of such tools are version control systems, issue trackers, build and test platforms and software usage monitoring platforms. Each of these has their own data formats and interfaces that makes combining the data into a holistic whole difficult. Because of this data is typically collected into a centralized data warehouse and homogenized for data processing needs. This paper focuses mostly on this architectural paradigm of centralizing preprocessed data and forms a basis for a data visualization and analysis system that can offer a holistic view into the software process.

Data architecture

The data architecture for a homogenized system has to be able to handle cases where the data coming in might be in multiple different for all the different artifacts and also handle different data velocities gracefully, as system monitoring data is typically updated on the second scale, while work item data is typically updated on a weekly cadence.

The data is also typically extracted from a multitude of different software applications, where the data representation for a work item changes between different platforms such as JIRA[1] and Azure DevOps[2].

In this paper we limit ourselves to certain datatypes directly related to software engineering. The data integrated into the software quality visualization system might however require data from the business area of the software developer to be added to these datatypes. For example eCommerce platforms might also want to homogenize sales data and link it to software development goals. The framework that we are underlining here is not limited to just the software engineering data artifacts, but uses them to focus the discussion.

Our system has to be able to handle this multitude of data sources for different types of software engineering data. Typical data types that are of interest in software engineering projects are work item data from issue trackers, Source code commit data from different version control systems, test reports from test systems, build and deployment reports from build systems, static analyzer results from code quality analysis and log data from usage monitoring platforms. These artifacts have different update frequencies outlined in Table 1.

[1] https://www.atlassian.com/software/jira

[2] https://azure.microsoft.com/en-gb/blog/introducing-azure-devops/

Table 1: Data update frequency

Data Type	Update Frequency, "Data resolution"
Work Items	Week / Day
Commit Data	Hour
Test reports	Hour
Build and Deployment logs	Hour
Static analysis	Hour/ Second
Server Usage Data	Second / Millisecond
Server Log Data	Millisecond

As can be seen from Table 1, the data we want to store from our data fetchers has a varying update frequency and data resolution for different types of artifacts. Selecting which resolution to save of your data is highly dependent on the update frequencies of the data you are integrating into your data warehouse. The data storage format has to be such that it is independent of the resolution of incoming data, as it would not make sense to store the state of the work items repository for each millisecond.

Another aspect of the data is the amount of linking between the different data types. In a typical software development environment you want to link the different data types to each other to infer relationships between bugs and commits for example, but this data is not always available.

Linking of the data at the source platforms is typically a process problem of the software pipeline under investigation. Many processes might be in place to, for example place the ID of the work item that a commit relates to - to the message field of a commit. One problem in creating a data warehouse of linked data is that this process format might vary from software pipeline to software pipeline and explicit links cannot always be trusted. As such ways to create implicit links after data collection can offer added value to the data fetching system.

The data fetchers should not lose any data in the artifacts that is present, which might be related to the linking of the artifact to other data artifacts. As such the data architecture should support the storing of arbitrary data fields on artifacts, as different software pipeline processes place emphasis on storing different information and different tools used might store metadata not present in others.

On top of this the data architecture in itself should support metadata added to the data artifacts after the data has been collected, to support the creation of explicit links between the different data artifacts. This also means that we should be able to form relationships between arbitrary data artifacts in the database to facilitate the storing of these links.

These scenarios outlined before give us a number of requirements for a data fetcher data architecture:

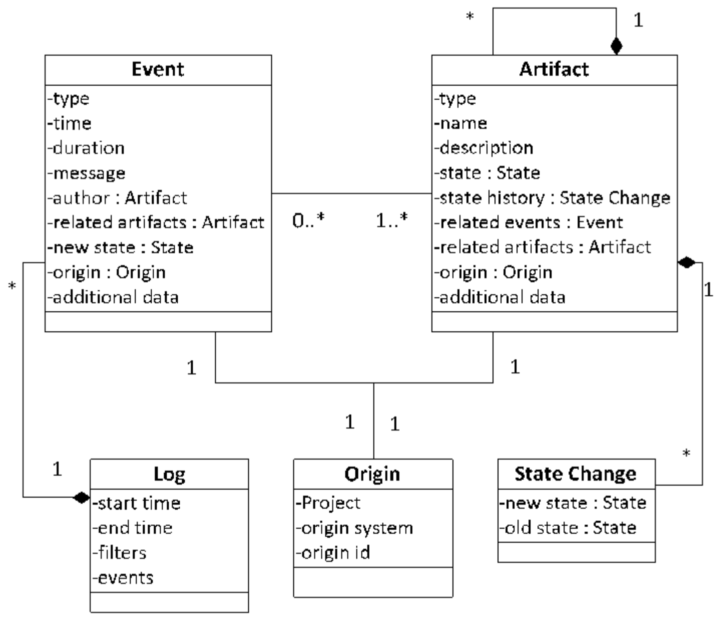
1. Data storage format has to be independent of the update cycle of a single data type

2. Each data type should be able to store varying data fields under the data types and the data fields might even vary under the same data type, based on the source system

3. The format should support the storage of metadata not originally fetched from the data source to facilitate the creation of implicit linking between data

4. The storage format should be able to handle relationships between arbitrary data objects in the data storage

Looking at the four requirements, the data architecture outlined requires a flexible schema structure, as all the data fields might not be known initially and to facilitate expansion of required fields in the database. These requirements are also slanted towards database solutions which are based on the “schema on read” paradigm, such as document databases and graph databases to fulfill the requirements outlined here. One such implementation of a data architecture can be found in the paper “Mashing up software issue management, development and usage data” by Mattila et al[3] and their approach is shown in Figure 1. This architectural model has also been implemented by Qentinel in the DevOps Quality Intelligence product.



[3] https://dl.acm.org/doi/10.5555/2820678.2820685

Figure 1 Data model from Mattila et al.

Data storage methods

Another consideration with the data architecture is the location of the database and storage architecture. Typically software engineering data is is considered core intellectual property by many companies and in some cases the access to the data can even be regulated by laws, for example GDPR for personal information. This places different restrictions to the physical architecture of the data fetchers.

Also the systems that you want to pull data from might be on-premise installations or on the cloud, available through public API's. For the purposes of this discussion, most of the systems with public API's available on the public internet do not differ from cloud sources in a major way. When we are referring to on-premises installations in this chapter, we refer o systems without a public API. The API can be available inside the customer's firewall however.

Typical data collection setups can be split into centralized cloud based setups and on-premise installations. The cloud based installations can also be split into single tenant and multitenant based applications. Both cloud formats share architectural features and also single tenant and on-premise installations share some architectural features.

In on-premise setups the data is stored locally in client premises and typically only accessible from there. But because the collection setup is also located inside the customers firewall, we can access all sensitive data there without it leaving customer premises. One option is also to pull data from cloud based services into the local installation, making it possible to combine the data there. The problems with this installation format are the typical software distribution problems. How to update and manage the on premises systems in multiple client organizations in an efficient way.

In cloud setups the data is stored in the cloud. With the two major branches being single-tenant installations into the cloud or multi-tenant installations into the cloud.

In single-tenant installations each customer has their own private sandbox in the cloud environment and data is stored in separate databases for each customer. Also the business logic layers for each client are handled separately in their own tenant. This solution has some of the drawbacks of the local setup, as each customer setup requires maintenance. It offers some control to the customer about their own environment and a perceived increase in security.

In multi-tenant cloud applications the data is stored in a single source, which is access control limited between the different tenants. The tenants share the business logic layers, which allows for more efficient handling of the data fetching layers if resources are shared between tenants. The multi-tenancy also allows easier comparison and contrasting of data between different tenants. This might enable new ways to share data between organizations and for example enable easier training of machine learning algorithms on top of the software engineering data.

In both cloud cases access to data in on-premises environments is limited, but data fetchers to cloud sources can be scaled easily. Cloud architecture also allows for the setup of both push and pull configurations of data fetching, where if the data warehouse in the cloud has a public inbound API, data can be with the right authentication pushed into the data warehouse. This push/pull configuration can also enable custom data preprocessing to be applied before the data is ingested by the data warehouse.

Data fetchers

Based on the physical architecture of your data collection platform, data fetchers can be roughly split into three categories, Local fetchers, Cloud fetchers and hybrid fetchers. Your collection platform can use a combination of these different types to access data located inside different subnets and enable filtering of the data, when crossing organizational or legal boundaries in the net. Typically these fetchers are created for a certain source or a datatype. Also typically we want to get the most up to date data into our data warehouse, which requires the pull services to be runnable on a schedule or watch mode.

Fetcher architecture

Typically a data fetcher refers to a piece of code, which accesses a data source, pulls data from it, preprocesses it and saves it into a data warehouse. This means we have to handle four different functions in a data fetcher: Access, data pulling, data processing and data saving.

These four tasks again give us a list of requirements for a data fetcher:

1. It must be able to access a data source system
2. It must be able to pull the data from the source system
3. It must be able to preprocess that data into a format accepted by the database
4. It must be able to save the data to a database

This gives us four architectural components, which we have to think of when designing a data fetcher.

Firstly, the fetcher has to be able to authenticate itself for the target data source, for example JIRA or Azure DevOps. This means that for data fetchers, we have to support multiple different login and authentication schemes. We also have to store login credentials in a secure way to facilitate easy login to these systems. In on premises setups this can be done inside the pull services, but in cloud infra a more robust system is needed.

Second, we must be able to pull the data from the source system. This requires a connector, which can access the API format offered by the data source and pull data. These formats might differ by source system, so we have to support multiple different data connection formats.

Third, when we have the data, we should be able to process that data into the homogenized data format required by our data warehouse. We also might have a raw data lake, where we indiscriminately dump the raw data also for backup purposes. This means that we have to have a way to convert data from multiple sources into a homogenized format.

Fourth, the now preprocessed data should be saved to a database. This is the most simple of the four tasks, requiring a single connection to the database.

With Mattila et al and at Qentinel we have arrived at a architecture for pull services where the data access and connection services are created per source system and these share a common data preprocessor for each type of data. The preprocessor then has a connector back to the database. This setup is outlined in Figure 2.

Fetcher deployment environments

Typically data fetchers can be installed in three different ways. They can be local, be in the cloud or a hybrid of these two we shall call hybrids.

Local fetchers are data fetchers installed in a local environment, where the data fetching is being done inside the firewall of a client. In actuality these do not differ a lot from cloud fetchers, with the exception that all the traffic that they generate stays inside the clients subnet and run on client provisioned infrastructure. These also limit the location of our data warehouse to the local environment, as they do not have access to the public net. Also scaling has to be done with the limits of customer infrastructure and this might also incur maintenance costs.

Cloud fetchers on the other hand are run in cloud provisioned infrastructure. They can be spun up when needed and depending on if we are running a multi-tenant or a single-tenant environment they can be limited to a single customer instance. Access credentials have to be handled by the data collection architecture and not just the single configured pull service. On the other hand this reduces maintenance and security risks, as these are managed services. The cloud fetchers have access to all source systems that have a public API.

The third data fetcher is a hybrid data fetcher that is a specialized version of a local fetcher. It is designed to permit either two way or one way communication to the cloud from inside the customer's firewall. Two way communication basically means that the local fetcher just has access to the public internet through a dedicated VPN connection for inbound communication. One way communication can also be set up in the hybrids, where the preprocessing logic and saving logic is changed. The preprocessing logic can be used to censor or remove critical information from the data it is sending to the data warehouse in the cloud and thus enable hybrid setups, where parts of the pipeline exist inside firewalled subnets, for example with mission critical code.

All the data fetchers described here can work be made to work with either push or pull configurations. If the data warehouse has a public API available, basically any script or service, which can authenticate to the warehouse and push data there can be considered a data fetcher. But for the purposes of this document, we consider a data fetcher to be a standalone entity, which by itself can access a target system, pull data from there to itself and push it to the datawarehouse. Thus the fetchers can be considered pull services for the data warehouse. This is clarified in Figure 3

Data fetcher management architectures

Last topic is the management of the configuration of the pull services. The management of the pull services can be made central or distributed and in the case of a central management, the management node can be located either in the cloud or in the on-premises environment.

When we are talking about a central management architecture, we are talking about a piece of software, which can create new pull services, observe the state of the pull services and shut them down when needed. This service can also store the login credentials for multiple different services and distribute them to new pull services as needed.

This can be achieved the easiest in the cloud, where using tools such as Kubernetes[4] or Amazon Fargate[5], the management system can create pull services when needed and pull new data into the store. This enables two behaviors for the system:

1. Batch mode, where the data is updated intermittently
2. Live mode, where pull services monitor changes in the data source system

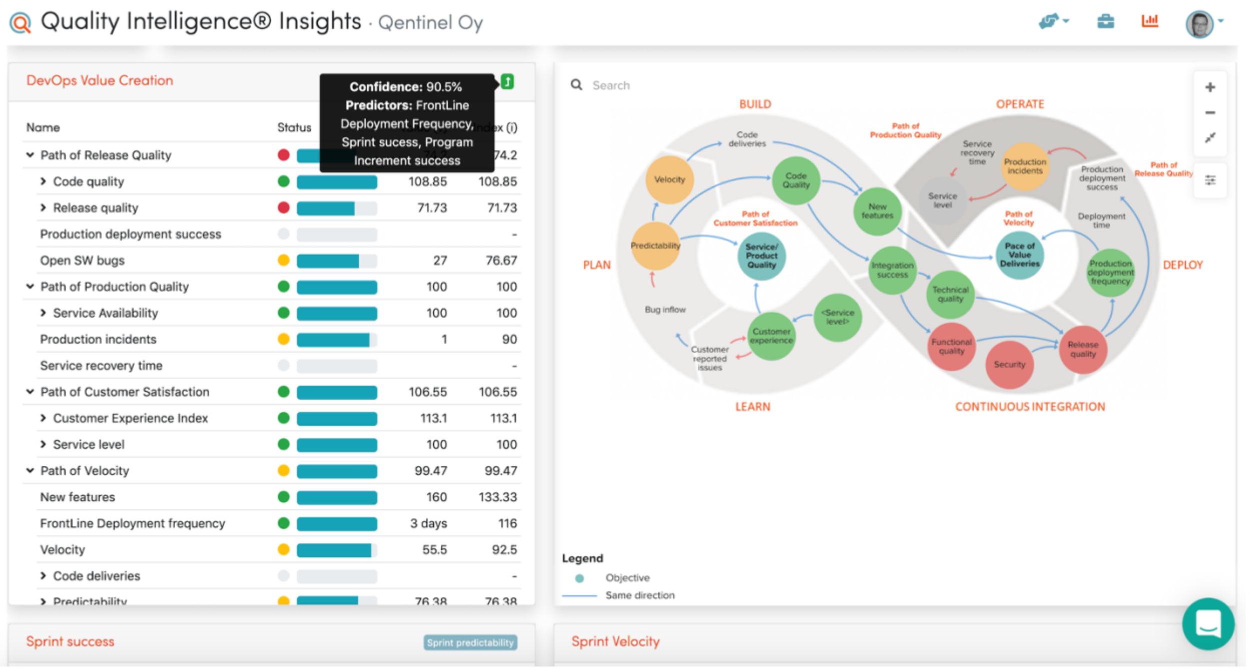
[4] https://kubernetes.io/

[5] https://aws.amazon.com/fargate/

The same system can also be deployed locally if a central system is wanted. Also using hybrid pull services with one way communication, we can get the cloud management services to be aware of the status of pull services inside firewalls, but we cannot directly control them. This hybrid setup enables you to get direct information from the cloud about the state of your system, enabling easier maintenance.

The system can also be managed in a distributed "unmanaged" way, where all the pull services are standalone processes, which just push data into the data warehouse. This way no central control exists, and data warehouse is reliant on external processes for its data update frequency.

Qentinel has built its data fetching architecture using a centralized cloud based approach. Our system can deploy pull services to the cloud infrastructure and manage them centrally from the cloud management. We also support hybrid pull services on premises with both two-way and one-way communication options for the client to choose from. The client can also censor data coming to the cloud from these pull services to be compliant with their data security processes. Qentinel uses this data fetching architecture to drive the Qentinel DevOps Quality intelligence dashboard illustrated in Image 1:



Quality Intelligence® Insights · Qentinel Oy

DevOps Value Creation

Confidence: 90.5%
Predictors: FrontLine Deployment Frequency, Sprint sucess, Program Increment success

Name	Status		Value (i)	Index (i)
⌄ Path of Release Quality	●		74.2	74.2
› Code quality	●		108.85	108.85
› Release quality	●		71.73	71.73
Production deployment success	●			-
Open SW bugs	●		27	76.67
⌄ Path of Production Quality	●		100	100
› Service Availability	●		100	100
Production incidents	●		1	90
Service recovery time	●			-
⌄ Path of Customer Satisfaction	●		106.55	106.55
› Customer Experience Index	●		113.1	113.1
› Service level	●		100	100
⌄ Path of Velocity	●		99.47	99.47
New features	●		160	133.33
FrontLine Deployment frequency	●		3 days	116
Velocity	●		55.5	92.5
› Code deliveries	●			-
› Predictability	●		76.38	76.38

Sprint success

Sprint predictability

Sprint Velocity

Image 1: Qentinel DevOps QI dashboard