Software Process MetaData (PMeta) Design Document

# Why PMeta

Modern software production systems are more than just a build server. It is a shifting mix of functions and technologies, from issue trackers and test frameworks to environment managers and repositories. At the same time, software supply chains become increasingly complex, crossing organizational and geographical borders many times before resulting in the final software product. It is a world of interconnected and interdependent software pipelines.

The PMeta framework applies the same architectural principles to this network of pipelines as to any other software design problem.

PMeta chooses open source Eiffel Protocol, enabling technology agnostic event-based communication among the actors of the system. Its implementation lets you know at a glance what is up in your software production system and your supply chain, while providing rich extension points where others can hook into your pipeline with perfect upstream and downstream traceability.

# What is PMeta

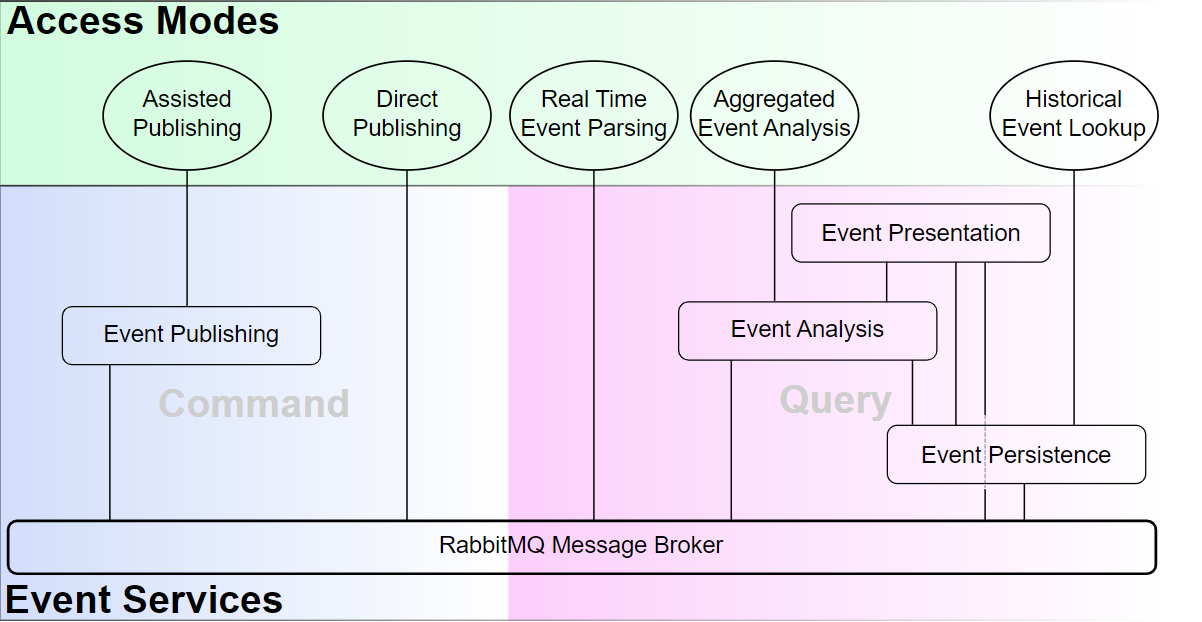
The PMeta is a framework for continuous integration and delivery, particularly addressing the challenges of a distributed and heterogeneous environment at an enterprise scale. It does this through the *in situ* real time generation of globally broadcast events - events which reference one another, forming a Directed Acyclic Graph (DAG) describing all the activities of the continuous integration and delivery pipeline, regardless of where they took place, which underlying technology they used, or even whether they were automated or not.

The PMeta framework consists of two parts. First, a vocabulary and syntax of events, forming the communication protocol of the framework. Second, services built on top of that communication protocol to orchestrate continuous integration and delivery activities, provide traceability, dashboards, visualizations and much more.

# Readers Notes

* Within the document, “Event” and “Message” are the same thing, a.k.a. message carrying software process metadata.
* Because the PMeta framework is based on the Eiffel Protocol, within the document, “PMeta Event” and “Eiffel Event” have the same meaning.

# PMeta Abstract Reference Architecture



## Event Services

An Event Service is a service that does not act in the pipeline, but exists to facilitate the handling of Eiffel events. This may be services for authoring (Command) or for consuming (Query) of such events. Event Services are typically not the provenance of Eiffel event data, but serve the Pipeline Actors and/or users by storing, analyzing, transporting and analyzing the event data.

### Event Analysis

The Event Analysis service enables the [Aggregated Event Analysis](#_8592tqeddpfh) mode of access. The service constructs new objects based on Eiffel event data. These objects may represent any type of entity, and typically reside at a higher level of abstraction than the Eiffel events.

It is worth noting that while Eiffel events are mutable, the objects constructed by an Event Analysis service may be mutable, depending on the individual implementation.

### Event Persistence

The Event Persistence service enables the [Historical Event Lookup](#_mybf7ksm5u03) mode of access. It stores events (all events received, or a filtered subset) as they are received via the [RabbitMQ Message Broker](#_dctn6vuxkqel), and exposes them for subsequent retrieval via an API.

While the internal storage and querying mechanisms, data models or underlying DBMS solutions of an Event Analysis service are irrelevant from Sepia's point of view, it is an absolute requirement for it to respect their immutability. In other words, any events retrieved from the Event Persistence service SHALL be exactly as received via the RabbitMQ Message Broke

### Event Presentation

The Event Presentation service presents event data, or data derived from events, in some format intended for human consumption. Often this involves some form of visualization of the data, although this is not necessarily the case. These presentations may be real time status displays (e.g. intended for information radiators), static on-demand displays or anything in between. Similarly, depending on the type of presentation, they may consume events from the [RabbitMQ Message Broker](#_dctn6vuxkqel) in real time, fetch historical events from an [Event Persistence service](#_e3sfw8hvs64d), fetch data aggregations from an [Event Analysis service](#_d2nhq323bjm2), or any combination of the three.

### Event Publishing

An Event Publishing service acts as a level of abstraction between authors of events and the [RabbitMQ Message Broker](#_dctn6vuxkqel), providing one or more of the benefits described in the [Assisted Publishing](#_qrkpldf1rea2) access mode. This is achieved via a REST API for authoring Eiffel events, as opposed to [Direct Publishing](#_27gd2gn8gd3y) via RabbitMQ client APIs.

### RabbitMQ Message Broker

The RabbitMQ Message Broker is the heart of the Sepia architecture. It routes all event messages from authors to recipients. The RabbitMQ Message Broker is implemented by [RabbitMQ](https://www.rabbitmq.com/). RabbitMQ is a widely deployed open source message broker and completely independent from the Eiffel community. In other words, there is no Eiffel specific message broker implementation. Instead, Sepia lists a small number of requirements on RabbitMQ servers and clients.

#### Server Requirements

The RabbitMQ Message Broker SHALL use RabbitMQ [version 3.0.0 or later](https://www.rabbitmq.com/changelog.html).

Note that Sepia places no requirements on deployment architecture (e.g. distributed or not), authentication enforcement or exchange configuration. These aspects of the message broker are up to the demands in each specific case.

#### Client Requirements

* Client implementations MAY use any [RabbitMQ Client API](https://www.rabbitmq.com/clients.html) compatible with the identified RabbitMQ server version (see Server Requirements).
* Client implementations SHALL support [RabbitMQ PLAIN authentication](https://www.rabbitmq.com/authentication.html).
* Client implementations MAY support [other types of authentication in addition to PLAIN](https://www.rabbitmq.com/authentication.html).
* Unless stated otherwise in their documentation, client implementations SHALL NOT make any assumptions with regards to server deployment or configuration.
* Unless stated otherwise in their documentation, client implementations SHALL NOT make any assumptions with regards to exchange topology, types or configurations.
* Unless stated otherwise in their documentation, client implementations SHALL NOT make any assumptions with regards to routing scheme.
* Unless stated otherwise in their documentation, client implementations SHALL NOT make any assumptions with regards to queue configurations.
* Any [Event Publishing](#_176kuja7myx8) service or pipeline actor implementation accessing the message broker via [Direct Publishing](#_27gd2gn8gd3y) SHALL act as a RabbitMQ producer publishing to a RabbitMQ exchange.
  + It SHALL ensure that published event types defined in edition-lyon of the Eiffel protocol comply with protocol definitions.
  + It MAY publish messages on Eiffel-like syntax, containing event types not defined in the Eiffel protocol, as well as other other types of messages completely unrelated to the Eiffel protocol.
  + It SHALL support routing keys.
  + It SHOULD use a routing key on the form ***eiffel.<family>.<type>.<tag>.<domainid>***, where:
    - ***family*** is the non-empty name of a group of Eiffel events to which the current event belongs. The families have not been defined in Sepia but may be in the future. Implementations may choose to use a fixed string in this field.
    - ***type*** is the type of the published Eiffel event (i.e. its meta.type member), e.g. EiffelArtifactCreatedEvent.
    - ***tag*** is an implementation-specific tag. It can be any non-empty string but must not contain a period.
    - ***domainid*** is the non-empty string representing the domain the event applies to. It corresponds to the meta.source.domainId member of an Eiffel event.
  + It SHALL support named exchanges.
  + Unless stated otherwise in its documentation, it SHALL NOT make any assumptions as to the presence, absence or nature of other producers connected to the same exchange.
* Any [Event Persistence](#_e3sfw8hvs64d) service, [Event Analysis](#_d2nhq323bjm2) service, [Event Presentation](#_5i69jz9vnufm) service or pipeline actor implementation accessing the message broker via [Real Time Event Parsing](#_x6ziihkl2i7y) SHALL act as a RabbitMQ consumer consuming from a RabbitMQ queue.
  + Unless otherwise stated in its documentation, it SHALL NOT make any assumptions as to the presence, absence or nature of other consumers connected to the same queue.

## Modes of Access

A Mode of Access captures a form of interaction with Eiffel events. It does not represent any physical entity, but merely describes a type of behavior.

### Aggregated Event Analysis

In Aggregated Event Analysis, the [Event Analysis](#_d2nhq323bjm2) service provides a layer of abstraction between the raw Eiffel events and their consumers. This is particularly useful in use cases where information contained in multiple events is required. While Eiffel events can be thought of as verbs communicating what is happening related to certain entities in the pipeline, [Aggregated Event Analysis](#_8592tqeddpfh) is about learning about those entities based on the sum of all those verbs.

Example: Whenever a new version of a release candidate has passed a certain set of test cases in a simulated environment, Jane wants to kick off tests in the target environment. She could do this by listening first to the *EiffelArtifactCreatedEvents* of the release candidates, and then listen to any *EiffelTestCaseFinishedEvents* referencing *EiffelTestCaseTriggeredEvents* referencing any of those release candidate artifact events. When any one *EiffelArtifactCreatedEvent* has been thus referenced by the correct set of downstream events, that means it's time to start the on-target test. This would require non-trivial logic, however, not to mention the need for storing the state of every release candidate and its downstream events. A much simpler way for Jane to achieve her trigger is to use an [Event Analysis](#_d2nhq323bjm2) service. This is because what she's interested in isn't really the events themselves, but the state of the release candidate: whenever a new release candidate has achieved the state of having passed the required test cases, Jane wants to be notified. This is easily achieved by setting up a subscription to the [Event Analysis](#_d2nhq323bjm2) service, freeing Jane's trigger from having to maintain any state.

### Assisted Publishing

In Assisted Publishing, actors do not publish events directly to the [RabbitMQ Message Broker](#_dctn6vuxkqel), but via an [Event Publishing](#_176kuja7myx8) service. While [Direct Publishing](#_27gd2gn8gd3y) does not require any additional services, Assisted Publishing offers several benefits:

* **No language specific library integration.** [Event Publishing](#_176kuja7myx8) services offer a language agnostic REST API that encapsulates the [RabbitMQ Message Broker](#_dctn6vuxkqel) integration.
* **Boilerplate generation.** Apart from the event type specific payload, all events contain a certain amount of boilerplate in their meta objects. [Event Publishing](#_176kuja7myx8) services can handle that on behalf of its clients, allowing them to focus on the content of the events they need to publish.
* **Link lookup.** Events reference one another via UUIDs, which means that historical events may sometimes have to be fetched from [Event Persistence](#_e3sfw8hvs64d) to create a fully formed new event. [Event Publishing](#_176kuja7myx8) services can perform this task on behalf of its clients.
* **Version negotiation.** Whereas staying up to date with the latest event type versions can be difficult for every single actor, [Event Publishing](#_176kuja7myx8) services can offer multiple API end-points for historical versions and create a corresponding event on the latest feasible version.

### Historical Event Lookup

Historical Event Lookup queries an [Event Persistence](#_e3sfw8hvs64d) service for events that have occurred in the past. This is required for any use case that operates not only in the present, but on information in past events.

### Direct Publishing

To publish Eiffel event data, actors may interface directly with the [RabbitMQ Message Broker](#_dctn6vuxkqel). When doing so, the actor takes full responsibility for behaving as a [RabbitMQ Client](https://www.rabbitmq.com/clients.html) and for authoring and serializing the JSON document constituting the event. For certain use cases this may be a preferable option, but the authoring and serialization of events can be non-trivial. In particular, the need to reference other events via UUID may require lookups using [Event Persistence](#_e3sfw8hvs64d). It is also worth noting that direct publishing requires library level RabbitMQ integration, whereas [Assisted Publishing](#_qrkpldf1rea2) offers REST API integration.

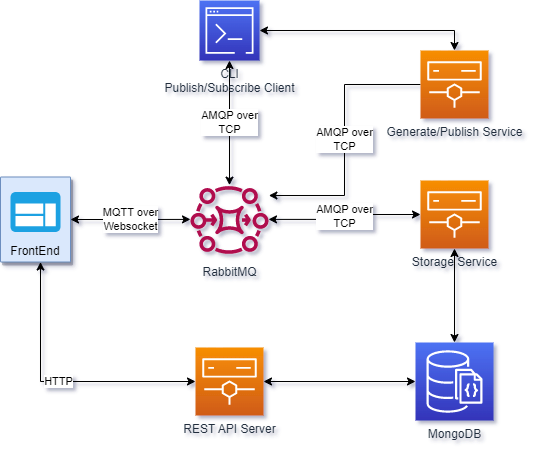
### Real Time Event Parsing

The most direct mode of consuming events is through real time parsing of events as they are received from the [RabbitMQ Message Broker](#_dctn6vuxkqel). This is done as a [RabbitMQ Client](https://www.rabbitmq.com/clients.html) consuming messages from a queue. This is a simple and quick method of accessing event data, but with limitations. While queues can be configured to retain messages received while the client is offline, thereby protecting against data loss due to downtime, any events that occurred before the queue was created will be inaccessible. Furthermore, the client will need to take responsibility for storing any event information needed for later use. In simple use cases, these limitations are not problematic. For instance, if all the information one needs is contained within a single Eiffel event, storing event data is not an issue.

Example: Jane needs to start a memory profiling job whenever a new release candidate is published. She listens to *EiffelArtifactPublishedEvent* matching the groupId and artifactId of her release candidates, and fetches the artifact at the location included in the event. This is an entirely stateless procedure that is easily implemented by consuming events directly from a RabbitMQ queue.

More advanced use cases, on the other hand, benefit from using [Historical Event Lookup](#_mybf7ksm5u03) and/or [Aggregated Event Analysis](#_8592tqeddpfh) instead.

# PMeta Fundamental Implementation Architecture



## RabbitMQ

See server Requirements in [RabbitMQ Message Broker](#_p29wtwvz11g).

## CLI

Users can publish and subscribe messages via CLI client directly.

CLI implementation follows [RabbitMQ Client Requirements](#_bjn3iaifs8z9)，[Direct Publishing](#_27gd2gn8gd3y), and [Real Time Event Parsing](#_x6ziihkl2i7y).

## Generate/Publish Service

Comparing against CLI client，this service provide REST API for generating, validating and publishing messages. It also maintains connections with RabbitMQ. It has below benefits,

* Massive users can share the same service without need to maintain connections with RabbitMQ, meanwhile reduce amount of connections to RabbitMQ servers.
* It provides programming language agnostic REST API, so it enables direct integration with any type of software pipeline to use the service.

The service implementation follows [Event Publishing](#_176kuja7myx8)和[Assisted Publishing](#_qrkpldf1rea2)

## Storage Service and REST API Server

Based on the MongoDB database, the Storage Service and REST API Server implements [Event Persistence Service](#_e3sfw8hvs64d).

* 1. Storage Service

Subscribe to messages and store them into the database, also it acts as an abstract layer between database and RabbitMQ message broker, making it easier to migrate to other databases in the future.

* 1. REST API Server

Query service is one of the key service in PMeta framework. The REST API Server provides services to search single event, downstream and upstream events and make queries based on complex combinations of conditions.

## FrontEnd（Web FrontEnd）

The FrontEnd implements [Event Presentation](#_5i69jz9vnufm) and [Real Time Event Parsing](#_x6ziihkl2i7y) in reference architecture.

RabbitMQ Message Broker needs to install and enable [Web MQTT Plugins](https://www.rabbitmq.com/web-mqtt.html)，so that the Web FrontEnd can listen to messages from RabbitMQ and visualize the events in real time.

# Overview of Selected PMeta Messages

## Introduction of Selected Messages

### Source Code Management Events

The Source Management events are responsible for storing and tracking source code revisions. In typical use cases, it is not triggered by information in Eiffel events, but rather by direct actions by developers (e.g. committing changes).

The **EiffelSourceChangeCreatedEvent** declares that a change to sources has been made, but not yet submitted.

The **EiffelSourceChangeSubmittedEvent** declares that a change has been integrated into to a shared source branch of interest (e.g. "master", "dev" or "mainline") as opposed to a private or local branch.

### Test Events

The **EiffelTestCaseTriggeredEvent** declares that the execution of a test case has been triggered, but not yet started.

The **EiffelTestCaseStartedEvent** declares that the execution of a test case has commenced. This event SHALL be preceded by a EiffelTestCaseTriggeredEvent, and appropriately linked to via TEST\_CASE\_EXECUTION.

The **EiffelTestCaseFinishedEvent** declares that a previously started test case (declared by EiffelTestCaseStartedEvent) has finished and reports the outcome.

### Artifact Events

The **EiffelArtifactCreatedEvent** declares that a software artifact has been created, what its coordinates are, what it contains and how it was created.

The **EiffelArtifactPublishedEvent** declares that a software artifact (declared by EiffelArtifactCreatedEvent) has been published and is consequently available for retrieval at one or more locations.

### Confidence Level Arbitration

The **EiffelConfidenceLevelModifiedEvent** declares that an entity has achieved (or failed to achieve) a certain level of confidence, or in a broader sense to annotate it as being applicable or relevant to a certain case (e.g. fit for release to a certain customer segment or having passed certain criteria). This is particularly useful for promoting various engineering artifacts, such as product revisions, through the continuous integration and delivery pipeline.

Confidence levels may operate at high or low levels of abstraction - ranging from "smokeTestsOk" to "releasable" or "released" - and they may group other confidence levels of lower abstraction levels. They may also be general or very niched, e.g. "releasable" or "reseabableToCustomerX". Confidence levels frequently figure in automated delivery interfaces within a tiered system context: lower level tiers issue an agreed confidence level signaling that a new version is ready for integration in a higher level tier.

### Composition Definition

The **EiffelCompositionDefinedEvent** declares a composition of items (artifacts, sources and other compositions) has been defined, typically with the purpose of enabling further downstream artifacts to be generated.

### Environment Provisioning

The **EiffelEnvironmentDefinedEvent** declares an environment which may be referenced from other events in order to secure traceability to the conditions under which an artifact was created or a test was executed. Depending on the technology domain, the nature of an environment varies greatly however: it may be a virtual image, a complete mechatronic system of millions of independent parts, or anything in between. Consequently, a concise yet complete and generic syntax for describing any environment is futile.

## Typical Relationships Between Events

