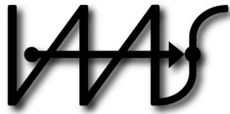


Institute of Architecture of Application Systems

Collaborative Gathering and Continuous Delivery of DevOps Solutions through Repositories

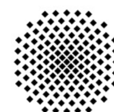
Johannes Wettinger, Uwe Breitenbücher,
Michael Falkenthal, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{wettinger, breitenbuecher, falkenthal, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@article{Wettinger2016,
  author    = {Johannes Wettinger and Uwe Breitenb{\"u}cher and
               Michael Falkenthal and Frank Leymann},
  title     = {Collaborative Gathering and Continuous Delivery of
               DevOps Solutions through Repositories},
  journal   = {Computer Science -- Research and Development},
  number    = {74},
  volume    = {22},
  year      = {2016}
}
```

© 2016 Springer-Verlag.
The original publication is available at http://www.springerlink.com
See LNCS website: http://www.springeronline.com/lncs



University of Stuttgart
Germany

Collaborative Gathering & Continuous Delivery of DevOps Solutions through Repositories

Johannes Wettinger, Uwe Breitenbücher,
Michael Falkenthal, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
{wettinger,breitenbuecher,falkenthal,leymann}@iaas.uni-stuttgart.de

Abstract Collaboration is a key aspect when establishing DevOps-oriented processes because diverse experts such as developers and operations personnel need to efficiently work together to deliver applications. For this purpose, highly automated continuous delivery pipelines are established, consisting of several stages and their corresponding application environments (development, test, production, etc.). The DevOps community provides a huge variety of tools and reusable artifacts (i.e. *DevOps solutions* such as deployment engines, configuration definitions, container images, etc.) to implement such application environments. This paper presents the concept of collaborative solution repositories, which are based on established software engineering practices. While this helps to systematically maintain and link diverse solutions, we further discuss how discovery and capturing of such solutions can be automated. To utilize this knowledge (made of linked DevOps solutions), we apply continuous delivery principles to create diverse knowledge base instances through corresponding pipelines. Finally, an integrated architecture is outlined and validated using a prototype implementation.

Keywords: Continuous Delivery, DevOps, Knowledge, Solution Repository

1 Introduction

DevOps [2,9,14] is an emerging paradigm, which aims to improve the collaboration between developers (‘dev’), operations personnel (‘ops’), and other parties involved in software development and delivery processes. Typically, cultural and organizational gaps between these groups appear, so different goals such as ‘push changes to production quickly’ on the development side versus ‘keep production stable’ on the operations side are followed. This often results in incompatible or even opposing processes and mindsets. *Continuous delivery* [8] is often used as a technical foundation to implement aligned DevOps-oriented processes in order to significantly shorten software release cycles. Especially users, customers, and other stakeholders in the fields of Cloud services, Web & mobile applications, and the Internet of Things expect quick responses to changing demands and occurring issues. Consequently, shortening the time to make new releases available becomes

a critical competitive advantage. In addition, tight feedback loops involving users and customers based on continuous delivery ensure building the 'right' software, which eventually improves customer satisfaction, shortens time to market, and reduces costs. An automated *continuous delivery pipeline* (also known as deployment pipeline) [8] is established to cover all required steps such as retrieving code from a repository, building packaged binaries, running tests, and deployment to production. Such an automated and integrated delivery pipeline improves software quality, e.g. by avoiding the deployment of changes that did not pass all tests. Moreover, the high degree of automation typically leads to significant cost reduction because the automated delivery process replaces most of the manual, time-consuming, and error-prone steps. Establishing a continuous delivery pipeline means implementing an individually tailored automation system, which considers the entire delivery process of a specific application. Along each pipeline with its different stages, corresponding application environments (development, test, production, etc.) are established.

The growing DevOps community provides a constantly increasing amount and variety of individual approaches such as tools and reusable artifacts to implement corresponding application environments. We refer to such tools and artifacts as *DevOps solutions* because they are used in the entire DevOps lifecycle[1]. Prominent examples are the Chef configuration management framework[2], the Jenkins[3] continuous integration server, and Docker[4] as an efficient container virtualization approach. The open-source communities affiliated with these tools publicly share reusable artifacts to package, deploy, and operate middleware and application components. However, such artifacts are stored and maintained in specific repositories such as Docker Hub, Chef Supermarket, and further repository platforms such as GitHub. Consequently, many solutions are isolated from each other. This leads to huge efforts to be spent in order to reuse such solutions seamlessly because users have to search in plenty of different repositories. This paper presents our work on a systematic approach to collaboratively gather diverse DevOps solutions in solution repositories in order to continuously deliver them in the form of different knowledge base instances. We partly build on a previously published DevOps knowledge management approach [13]. In particular, the major contributions of this paper can be summarized by (i) the concept of *collaborative solution repositories*, (ii) an *automated discovery and capturing approach* to populate these solution repositories, (iii) the concept of *continuously deliver knowledge base instances* based on the solution repositories, and (iv) the GatherBase *architecture* in conjunction with a prototype implementation to evaluate the presented approaches.

[1] DevOps lifecycle: `http://newrelic.com/devops/lifecycle`
[2] Chef: `http://www.chef.io`
[3] Jenkins: `http://jenkins-ci.org`
[4] Docker: `http://www.docker.com`

2 Collaborative Solution Repositories

Collaboration is a key aspect when implementing DevOps practices and continuous delivery in particular because diverse experts such as developers and operations personnel are involved and thus need to collaborate. A broadly established approach to enable collaboration in software development and operations are shared repositories. Developers typically collaborate through code repositories based on established version control systems such as Git or Subversion. Beside the source code of an application, such repositories contain tests, build scripts, deployment plans, and further supplementary materials such as documentation. Moreover, operations personnel maintains configurations through such repositories, often in the form of structured documents using markup languages such as XML, YAML, or JSON. Version-controlled repositories provide a high degree of provenance and transparency because all participants can browse and follow the history of a repository with all its changes. This does not only make collaboration more effective, but also establishes a trustful environment because also the smallest change is visible to all participants.

Typically, experts (developers, operations personnel, etc.) are familiar with the approach of maintaining structured documents inside such repositories. While experts are also familiar with the associated tools and languages such as Git, XML, JSON, etc., the approach itself is typically based on standards (e.g. XML or JSON) and de-facto standards (e.g. Git), so there is no strict vendor lock-in. Consequently, each and every expert uses his favorite tooling (e.g. any Git client and an XML editor) to effectively participate in collaborative processes, which are centered around such repositories. Diverse participants can collaborate in a decoupled manner because each of them owns a high degree of freedom when building his local environment. This approach turned out to be successful in the field of software development and operations, both inside and outside of organizations. GitHub[5] is a prominent example for a widely adopted open-source platform and community, which is completely centered around repositories. Beside basic repositories, such platforms typically provide feature-rich collaboration techniques such as pull requests[6] as well as collaborative review and discussion capabilities.

Therefore, we adapt and apply this established approach to maintain solutions and their metadata as structured documents inside version-controlled repositories. The major goal is to use collaborative *solution & metadata repositories* (in short *solution repositories*) as a foundation to establish and maintain a consolidated knowledge base, which is made of many different solutions and their metadata. Instead of the conventional idea of centering repositories around specific software applications, a solution repository stores the metadata describing reusable solutions and the solutions themselves (or references to these); these solutions are used as building blocks of diverse application environments (development, test, production, etc.), e.g. to implement continuous delivery pipelines. The previously

[5] GitHub: `http://www.github.com`

[6] Pull Requests: `http://help.github.com/articles/using-pull-requests`

discussed aspects and success factors of repositories (standards-driven, free choice of tooling, etc.) equally apply to solution repositories. As a result, diverse experts can participate in collaborative processes to maintain and utilize corresponding, mostly application-agnostic solutions through a knowledge base, which is based on solution repositories. A solution is made of the following parts, which are specified through its metadata:

- A URI, which uniquely identifies the solution.
- A set of labels to specify the solution's capabilities.
- A set of links to other solutions or labels (i.e. the 'solution boundary') to express requirements, conflicts, and recommendations.
- Arbitrary properties (key-value pairs) to further characterize the solution and attach additional content such as parameters, files, references, etc.

In order to represent such solutions and their metadata as structured documents in repositories in a normalized form, established markup languages such as XML, JSON, YAML, or Markdown are utilized. Schema definitions can be utilized to define a concrete serialization for specific languages such as JSON schema or XML schema. The following listing outlines an example for representing metadata of a solution as structured document using JSON:

```
1  {
2    "url": "https://supermarket.chef.io/cookbooks/apache2/versions/3.0.0",
3    "labels": [
4      { "is": "Executable/Script/Chef Cookbook" },
5      { "provides": "Middleware/Web Server/Apache HTTP Server" }
6    ],
7    "links": [
8      { "requires_host": "Infrastructure/Operating System/Linux/Debian" },
9      ...
10   ],
11   "properties": {
12     "name": "apache2 Chef cookbook",
13     "revision": "3.0.0",
14     "description": "Installs and configures all aspects of apache2 ...",
15     "maintainer": { "name": "svanzoest", "email": "..." },
16     "license": "Apache 2.0",
17     "info_url": "https://supermarket.chef.io/cookbooks/apache2",
18     "package_url": "https://supermarket.chef.io/api/v1/cookbooks/apache2
           /.../3.0.0/download",
19     "repository_url": "https://github.com/svanzoest-cookbooks/apache2",
20     "readme": "...",
21     "chef_recipes": {
22       "apache2": "Main Apache configuration",
23       "apache2::logrotate": "Rotate apache2 logs",
24       "apache2::mod_alias": "Apache module 'alias' with config file",
25       ...
26     },
27     ...
28   }
29 }
```

However, this is just a single alternative how to represent a solution using JSON. Another example based on XML is available on GitHub[7]. Similar to application source code, these structured documents can be collaboratively maintained and shared using version-controlled repositories, i.e. collaborative

[7] XML sample: `https://gist.github.com/jojow/03f368aad0326273e8b5`

solution & metadata repositories. As discussed previously, diverse experts can use their favorite tooling to create, edit, and update these documents. Labels are a key concept to classify and link solutions among each other. On the one hand, labels are utilized to specify characteristics (e.g. being an executable of type *Chef Cookbook*) and capabilities (e.g. providing an *Apache HTTP Server*) of a solution. On the other hand, links can refer to labels instead of solutions, e.g. to express requirements that can be satisfied by different solutions, providing corresponding capabilities (e.g. providing a *Debian* operating system). These labels are organized using a label taxonomy, i.e. a label hierarchy. The taxonomy is not static, but needs to be changed if, for instance, new categories of solutions appear, or existing categories of solutions are refined. Therefore, an ongoing and collaborative *taxonomy evolution* is happening. For this reason the label taxonomy is also maintained through structured documents inside solution repositories. The following listing shows an example for the middleware dimension of the label taxonomy, rendered using YAML:

```
Middleware:
  Runtime:
    properties:
      alias:
      - $parent/Runtime Environment
    Java:
      properties:
        alias:
        - $parent/Java Runtime Environment
        - $parent/JRE
      Scala:
      Groovy:
        Grails:
    Python:
      Jython:
        properties:
          requires:
          - Middleware/Runtime/JRE
    Ruby:
      JRuby:
        properties:
          requires:
          - Middleware/Runtime/JRE
    PHP:
    Node.js:
    ...

  Web Server:
    Nginx:
    Apache HTTP Server:
      properties:
        alias:
        - $parent/Apache HTTPD

  Data Store:
    properties:
      alias:
      - $parent/Database
    Object Store:
    In-Memory:
    Relational:
      MySQL:
      MariaDB:
      PostgreSQL:
    Document-oriented:
```

```
46        MongoDB:
47        CouchDB:
48      Key-Value:
49        Redis:
50          properties:
51            alias:
52            - Middleware/Cache/Redis
53            - Middleware/Data Store/In-Memory/Redis
54            - Middleware/Messaging/Redis
55        Riak:
56        Cassandra:
57      ...
```

Beside the label hierarchy, the taxonomy contains alias labels (e.g. *Apache HTTPD* is an alias for *Apache HTTP Server*), which can be used alternatively to their primary labels. This is to cover and map different naming conventions and established terms used by different communities. Furthermore, requirements can be attached as properties to a label. This makes sense in case all solutions that are associated with a particular label inherently own a specific requirement. For example, all solutions providing *JRuby*[8] require a *Java Runtime Environment*.

To sum up, solution repositories in conjunction with structured documents enable effective collaboration among diverse experts. The entire approach is based on established practices and tooling how software developers and operations personnel collaborate in modern environments. However, fully manually maintaining all parts of these repositories does not scale due to the huge amount and variety of solutions. Especially the solutions' metadata quickly become outdated because solutions could be developed by different people than the maintainers of solution repositories. To tackle this issue and to simplify the maintenance of solution repositories, the following Sect. 3 presents an approach to automate the discovery of certain solutions and store them in solution repositories. Furthermore, there is another challenge that needs to be addressed, namely how to query and efficiently utilize the knowledge base, which is based on the previously discussed solution repositories. While managing structured documents stored in potentially distributed solution repositories nicely works for collaboratively maintaining the knowledge base, these repositories typically do not provide fine-grained query mechanisms to utilize the knowledge base in an efficient manner. Moreover, consistency checks are not made, e.g. by verifying whether the given documents are structured properly and the utilized labels actually comply to the label taxonomy. Therefore, Sect. 4 presents an automated approach (i) to check the involved solution repositories for consistency, and (ii) to generate a consolidated instance of the knowledge base, providing a query interface.

3 Automated Discovery and Capturing of Solutions

Figure 1 presents an overview of the *auto-gather*[9] pipeline to automate the discovery and capturing of solutions. The pipeline consists of five stages: (i) during

---

[8] JRuby: `http://jruby.org`

[9] Automated gathering of solutions

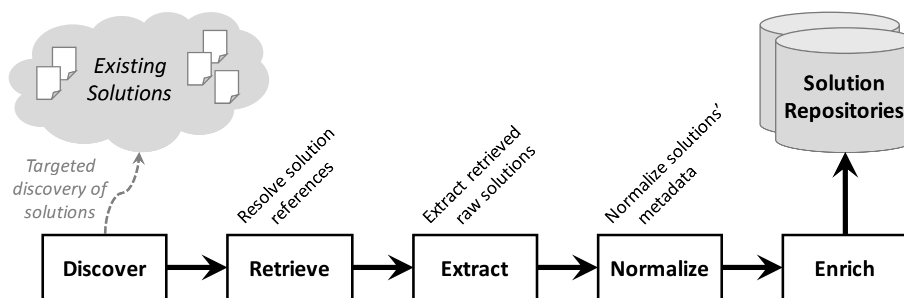


Figure 1. Overview of *auto-gather* pipeline

discover stage, existing solutions from different sources such as Chef Supermarket[10] and Docker Hub[11] are identified. (ii) The *retrieve* stage consumes solution references, mostly URLs, which are produced by the *discover* stage. These references are then resolved by retrieving the raw solutions. The retrieval can happen through diverse channels such as HTTP, Git, Bazaar, Rsync, etc., depending on where a particular solution is located. (iii) During *extract* stage, relevant metadata are extracted and derived from previously retrieved raw solutions. (iv) The *normalize* stage aligns their representation with the specified metadata representation, e.g. based on JSON documents as discussed in Sect. 2. (v) Finally, during *enrich* stage, metadata are refined, e.g. by applying document classification techniques [12] for assigning labels to solutions to better characterize their capabilities. A simple classification approach would be to match keywords between the label taxonomy and the metadata of a solution. These normalized and enriched metadata are eventually stored in solution repositories. Obviously, the solution repositories may be accessed during *enrich* stage to retrieve the current label taxonomy, which enables the assignment of additional labels to retrieved solutions for classification purposes.

The *auto-gather* pipeline is not meant to replace the mostly manual collaborative approach based on solution repositories, which was presented in Sect. 2. The two approaches are rather complementary, so the *auto-gather* pipeline populates solution repositories in an automated manner, while diverse experts are still able to collaboratively maintain additional or refine existing solutions. This hybrid approach is a significant improvement over just manually maintaining solution repositories as discussed in Sect. 2: the automated gathering of solutions makes the entire approach scale much better because large amounts of solutions can be automatically discovered and captured.

Although the solution repositories, partly populated automatically and manually, provide a collaborative foundation for a comprehensive knowledge base, there are still two major issues: (i) the solutions stored inside the repositories are not checked for consistency, e.g. whether their representation complies with a given

[10] Chef Supermarket: `https://supermarket.chef.io`
[11] Docker Hub: `https://hub.docker.com`

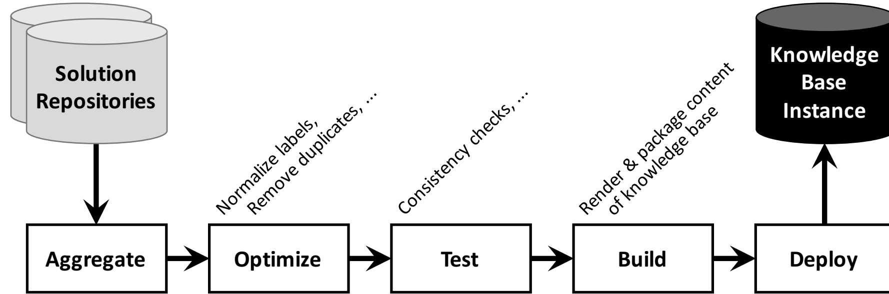


Figure 2. Overview of *deliver-kb* pipeline

schema or the utilized labels are actually valid regarding the label taxonomy. (ii) Fine-grained query mechanisms are missing; however, these are required to utilize the knowledge base in an efficient manner. To tackle these issues, Sect. 4 presents an automated approach for performing consistency checks and generating consolidated instances of the knowledge base with a proper query interface.

4 Continuous Delivery of the Knowledge Base

As discussed previously, solution repositories enable the systematic gathering and maintenance of diverse solutions and their metadata in a collaborative manner. However, solutions and metadata inside such repositories are simply represented as file-based structured documents as explained in Sect. 2. This approach is similar to managing application-specific source code files through such repositories. Consequently, neither consistency checks are performed when adding or modifying files, nor fine-grained query mechanisms are provided to find and identify appropriate solutions for a certain scenario. These deficiencies are tackled by adapting established continuous integration and continuous delivery practices [8], because they solve similar issues when dealing with source code repositories for specific software applications. As an example, a continuous delivery pipeline runs various tests to check the correctness and consistency of newly committed and modified source code. Moreover, such a pipeline continuously delivers updated instances of the corresponding application through automated builds and deployments. Figure 2 outlines an overview of the *deliver-kb*[12] pipeline, which makes use of these key concepts to continuously deliver updated instances of the knowledge base. With this approach, the 'eat your own cooking' principle is applied by utilizing and adapting the key concepts of continuous delivery pipelines to deliver instances of a knowledge base, which themselves are used to implement continuous delivery pipelines for specific applications.

The stages of the *deliver-kb* pipeline are similar to the various stages of common continuous delivery pipelines. However, they are adapted to fit the needs

[12] Deliver knowledge base

of implementing continuous delivery of a knowledge base instead of a specific software application. For this purpose the initial *aggregate* stage consolidates solutions captured in potentially multiple solution repositories. Reasons why solutions may be distributed across different solution repositories can be diverse: beside plain separation of concerns, some solutions may be private, while others are public or at least shared among several organizations. Optionally, filters could be applied in this stage, e.g. to exclude certain solutions, which should not be part of the resulting knowledge base. During *optimize* stage, the aggregated set of solutions can be refined in various ways. For example, duplicate solutions can be eliminated to avoid polluting the knowledge base. Furthermore, the labels describing requirements and capabilities of solutions can be normalized by replacing alias labels by their primary ones. The *test* stage covers the previously mentioned consistency checks. This may include schema-based validation of solutions' metadata as well as checking whether the utilized labels are valid regarding the label taxonomy. Then, the linked set of solutions are rendered and packaged as content of the knowledge base during *build* stage. This is similar to building the binaries of an application, which can then be deployed in the next step, i.e. during *deploy* stage.

The deployment eventually results in concrete instances of the knowledge base. These instances, which are created through the *deliver-kb* pipeline, provide query mechanisms to process requests and produce responses, containing or pointing to appropriate solutions. The technical foundation of such a knowledge base instance can be diverse. To make a few examples, an instance could be provided as (i) a relational database in conjunction with a REST API, (ii) a Web-based GUI with a full-text search engine as back-end, (iii) a graph database with a corresponding query API, or (iv) an RDF store with a SPARQL query interface. Multiple *deliver-kb* pipelines can be established to deploy different kinds of knowledge base instances, which may be targeted to different groups of users. Similar to deploying software applications to different environments (development, test, production, etc.), knowledge base instances can be deployed to different environments, too. For example, a developer can run a minimal instance locally on his developer machine, whereas an organization may run a full-blown and scaled out instance in its private cloud environment.

The presented approach of continuous delivery of diverse knowledge base instances, backed by collaborative solution repositories is based on several concepts, which are successfully established to collaboratively develop and operate applications. Automated continuous delivery pipelines and source code repositories are two key building blocks in this context. However, any knowledge base naturally owns the risk of getting outdated, especially if the covered scope of knowledge changes quickly. The knowledge base discussed in this work is based on a huge variety of DevOps solutions maintained through solution repositories. These solutions provide building blocks for establishing continuous delivery pipelines as well as implementing deployment automation. Since the DevOps community is not only growing, but also very active and disruptive, it is a challenge to keep up with constantly updating and newly emerging solutions. Consequently, the risk

of an outdated knowledge base is a real problem in this context. The presented approach tackles this risk with two approaches: (i) the automated discovery and capturing of solutions (Fig. 1) keeps at least parts of the underlying solution repositories updated. (ii) By creating knowledge base instances through automated delivery pipelines (Fig. 2), these instances are always built and deployed based on the latest revisions of the involved solution repositories. This helps to avoid the creation and usage of instances, which provide an outdated set of solutions. However, these two approaches do not help to keep the manually maintained parts of the solution repositories and generated knowledge base instances updated. This risk is mitigated by the significant overlap between potential producers and consumers of the knowledge base. Developers, operations personnel, and further experts utilize, i.e. consume the knowledge base, but they also produce contents and add it to the knowledge base in the form of updated and newly added solutions. This overlap keeps users of the knowledge base motivated to also contribute contents to the underlying repositories because they are immediately interested in keeping the quality of the resulting knowledge base high. Otherwise the knowledge base becomes less usable for them over time. Another motivation for contributing solutions may be the fact that such solutions increase their visibility to foster reuse by other experts. With this approach, the knowledge base and the underlying shared solution repositories foster collaboration between different kinds of experts, thereby implicitly supporting recently emerging software development paradigms such as DevOps [2]. As discussed previously, the presented approach utilizes established concepts and tooling such as structured documents maintained through version-controlled repositories and automated delivery pipelines. Consequently, the barrier for contributing to solution repositories and the resulting knowledge base is relatively low for potential users such as developers and operations personnel. This fact also helps to lower the actual risk of an outdated knowledge base.

5 GatherBase Architecture & Prototype Implementation

The previous sections discussed several key concepts, which are required to achieve collaborative and automated gathering of solutions. The presented concepts are described in an abstract manner, so they can be applied and implemented in various ways. This section aims to present an integrated architecture, namely the GatherBase architecture, which covers all previously discussed concepts from *gather*ing solutions in solution repositories to creating diverse instances of the knowledge *base*. Figure 3 shows an overview of the GatherBase architecture, which is fully plugin-based to be highly modular and extensible. The upper part (above solution repositories) covers the *auto-gather* pipeline described in Sect. 3 for the automated gathering of solutions. Three kinds of job processors connected by corresponding job queues provide a loosely coupled architecture, covering the stages of the *auto-gather* pipeline. Specialized discovery job processors cover the *discover* stage. As an example, the *Chef cookbook discoverer* performs a targeted discovery of Chef cookbooks, which are provided through the Chef Supermarket.

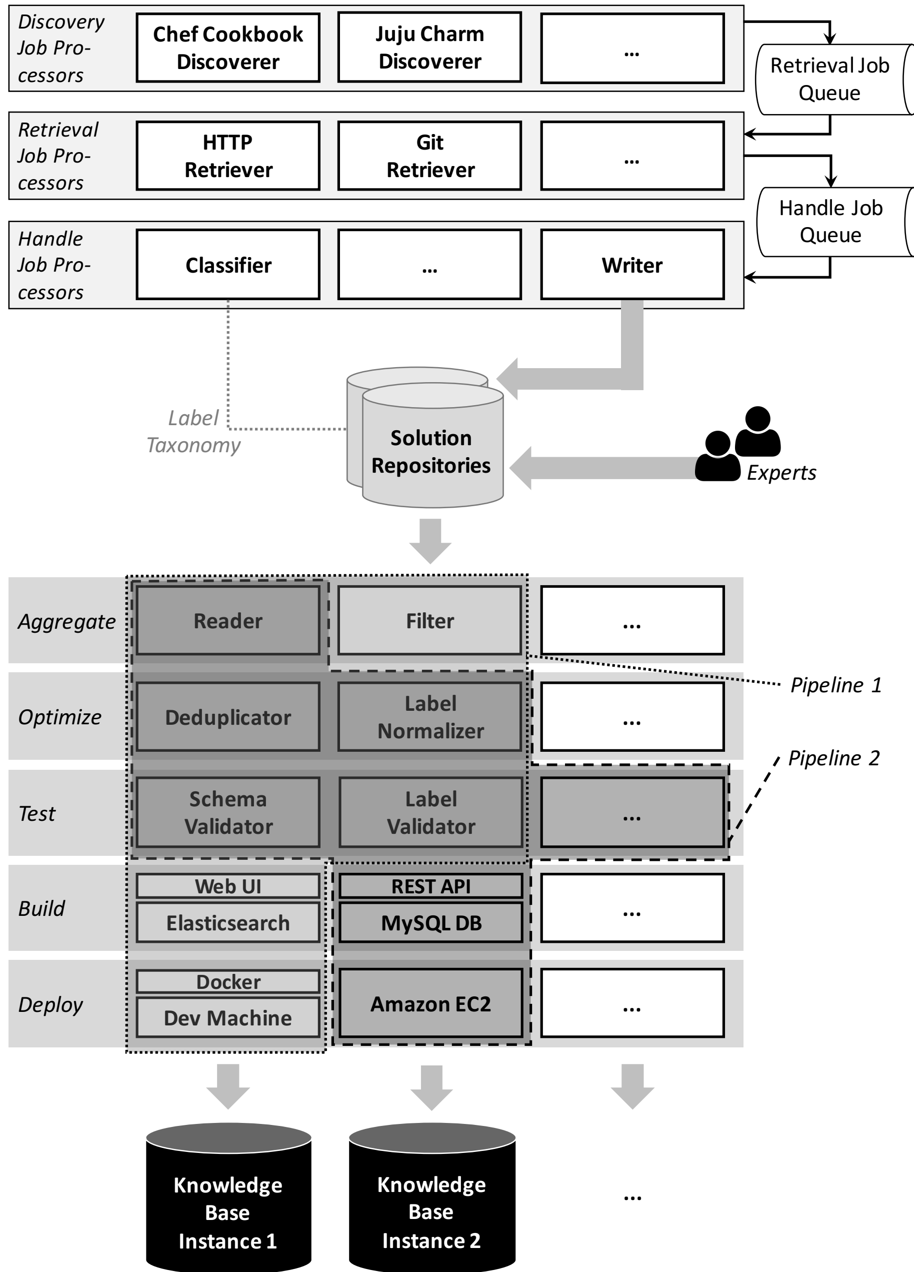


Figure 3. Overview of GatherBase architecture

The discovery may be triggered in various ways. A straightforward approach would be to run each discoverer periodically at certain time intervals to check for updated and newly added solutions. Alternatively, an event-based approach may be followed, e.g. by a discoverer subscribing to a specific source, which notifies the discoverer about changes. Consequently, the corresponding discoverer can react to incoming change notifications instead of periodically polling for updates.

Each discoverer produces a separate retrieval job for each discovered artifact or solution and puts the job into the *retrieval job queue*. Jobs in this queue are asynchronously consumed by retrieval job processors, which cover the *retrieve* stage of the pipeline. Depending on where a specific solution is stored, a corresponding retriever is invoked. If, for instance, a particular Chef cookbook is stored in a Git repository, the *Git retriever* is used to fetch the cookbook.

For each successfully retrieved solution, a separate handle job is put into the *handle job queue*. This queue is asynchronously consumed by diverse handle job processors, in short *handlers*, which cover the stages *extract*, *normalize*, and *enrich*. Some handlers are run for all solutions, others are only run for specific kinds of solutions. As an example, a *ZIP file handler* is utilized to extract the contents of a solution during *extract* stage. Then, a *Chef metadata handler* may be used (specifically for Chef cookbooks only) to transform Chef-specific metadata into a normalized representation during *normalize* stage. The *classifier* uses the label taxonomy to categorize a given solution by adding labels to the solution to specify its capabilities. This happens during *enrich* stage. Finally, the *writer* puts the normalized and enriched solution metadata – bundled with the solution itself or a reference to it – into a solution repository. To determine which handlers are run for a specific handle job, a set of rules is defined as configuration to be evaluated at runtime. As a result, an individual chain of handlers is dynamically identified at runtime for each handle job. For instance, a simplified handler chain for a Chef cookbook may be as follows: *ZIP file handler* → *Chef metadata handler* → *classifier* → *writer*.

The middle part of Fig. 3 positions the solution repositories as link between *auto-gather* pipeline building blocks (upper part) and *deliver-kb* pipeline building blocks (lower part). Beside solutions and their metadata, these repositories contain the label taxonomy, which is, for example, used by the *classifier* to categorize solutions according to the taxonomy. In addition to the *auto-gather* pipeline, solutions in repositories can be maintained and refined by experts manually.

The lower part (below solution repositories) covers the *deliver-kb* pipeline described in Sect. 4 for the creation of concrete knowledge base instances. For each stage, a set of modules is provided, which are used to establish different variants of the pipeline, depending on the kind of knowledge base instance that should be created. Figure 3 outlines two example pipelines that can be implemented by combining different sets of modules across the pipeline stages to eventually create diverse knowledge base instances. *Pipeline 1* uses *reader* and *filter* during *aggregate* stage to fetch selected solutions from the solution repositories. The *deduplicator* is utilized during *optimize* stage to remove duplicate solutions. Moreover, the *label normalizer* replaces alias labels by primary labels

according to the label taxonomy. Next, during *test* stage, the *schema validator* checks whether the solutions' metadata are represented properly according to given schema definitions such as an XML schema definition for solution metadata documents, which are expressed using XML. The *label validator* checks the utilized labels (for expressing requirements and capabilities of solutions) against the label taxonomy to ensure whether they are valid labels. Once the *test* stage finished successfully, the knowledge base instance is built during *build* stage. In case of *pipeline 1*, an *Elasticsearch*[13] instance is populated with the contents of the knowledge base, providing the back-end of the knowledge base instance. Moreover, a *Web UI* is packaged as front-end together with the populated *Elasticsearch* back-end to ease the interaction with the knowledge base instance. Finally, the packaged instance is deployed using *Docker* on a *developer's machine* during *deploy* stage.

Pipeline 2 outlines a second example: the stages *aggregate*, *optimize*, and *test* are pretty similar to *pipeline 1* in terms of the utilized modules. However, the *filter* is omitted in this case, i.e. all solutions stored in the repositories are considered. The *build* and *deploy* stages are completely different: in this case a *MySQL database* instance is populated with the contents of the knowledge base. A *REST API* is packaged together with the *MySQL* back-end to provide a Web-based query interface for the underlying knowledge base. This knowledge base instance is then deployed to *Amazon EC2*[14] to run as a cloud-based service. Many other variants of the pipeline could be established using this architecture. Such a pipeline can then either be triggered on each commit to one of the solution repositories, or it can be triggered on demand or periodically at certain time intervals.

Our prototype implementation is based on a set of modules, which are implemented using Node.js. These modules are integrated to implement *pipeline 2* that is outlined in Fig. 3. Resulting knowledge base instances, which are produced by this pipeline, provide an Elasticsearch back-end that can be queried using Elasticsearch's JSON-based domain-specific query language[15].

6 Related Work

As outlined previously, our presented approach to collaboratively gather and continuously deliver DevOps solutions through repositories is based on several established software engineering concepts. These include collaborative repositories and continuous delivery pipelines [8] for software applications. Beside these basic concepts, the automated discovery and capturing of solutions as discussed in Sect. 3 utilizes concepts and tooling from general-purpose Web crawling [7,3]. Our presented approach differs from general-purpose crawlers because it aims for targeted gathering of specific kinds of solutions. Consequently, the discovery and capturing components are much more specialized, so the quality of the results is

[13] Elasticsearch: `https://www.elastic.co/products/elasticsearch`
[14] Amazon EC2: `https://aws.amazon.com/ec2`
[15] Sample query: `https://gist.github.com/jojow/edb262290d1acb406e36`

higher. However, technically, a lot of established Web crawling-related techniques, libraries, and other tooling can be reused to build specialized components for discovering and capturing solutions.

Furthermore, domain-specific knowledge management approaches appear in various fields. Architectural knowledge management [1,5,11] is a prominent example to enable the systematic capturing and reuse of architecturally relevant knowledge, which helps to build and refactor software applications. Ontology-based knowledge engineering and its relationship to software engineering [10] is another related approach. While some of these approaches aim to promote specific tooling to establish and use a domain-specific knowledge base, our presented approach focuses on decoupling knowledge creation and maintenance (through solution repositories) from knowledge utilization (through continuously delivered knowledge base instances). This allows for creating different kinds of knowledge base instances for diverse use cases. Moreover, the underlying concepts are well established in software development and operations, so experts do not have to get familiar with novel tooling or workflows: they can reuse their favorite tooling such as their Git client, JSON editor, etc. to create and maintain knowledge similarly to creating and maintaining source code. 'Knowledge as code' is coined as a term in this context to emphasize that knowledge should be treated similar to regular source code. Beside our presented approach, further modern practices to knowledge management exist, which follow the notion of 'knowledge as code', such as managing structured Markdown documents in Git repositories[16].

The concept of solution repositories was previously introduced in a different context, namely in the domain of pattern research [6]. While in this field a solution repository is used to store and manage concrete artifacts that implement abstract patterns [4,6], our approach also comprises knowledge artifacts such as fine-grained documentation and technical manuals, which are not immediately executable.

7 Conclusions

The previously presented GatherBase architecture provides a comprehensive approach to implement the collaborative and automated gathering of solutions as discussed in this paper. As a key prerequisite and enabler for this approach, we presented the concept of collaborative solution repositories (Sect. 2), which are based on established practices how software developers and other experts efficiently collaborate through repositories. Moreover, we discussed an automated discovery and capturing approach to populate these solution repositories (Sect. 3) as another key concept to make the entire approach scale much better. The third fundamental building block is continuous delivery of knowledge base instances based on the solution repositories (Sect. 4). Finally, we evaluated these concepts in an integrated manner through the discussed GatherBase architecture and its prototype implementation (Sect. 5).

[16] `https://www.cloudbees.com/blog/knowledge-code-sourcing-your-knowledge`

In terms of future work we aim to extend the scope of the presented approach. The GatherBase architecture as well as its underlying concepts are domain-independent. Therefore, the presented architecture can be adapted in future work to gather and reuse solutions in other domains. Examples may include solutions to implement business processes or solutions to build and connect devices as part of the Internet of Things. Moreover, we aim to establish a rich ecosystem of modules to be used for implementing continuous delivery pipelines for diverse knowledge base instances. This is then the foundation for the next step, namely to systematically evaluate which kinds of knowledge base instances are suitable for which usage scenarios.

Acknowledgments This work was partially funded by the DFG project SitOPT (610872) and the BMWi project SmartOrchestra.

References

1. Babar, M.A., Dingsøyr, T., Lago, P., van Vliet, H.: Software architecture knowledge management. Springer (2009)
2. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. SEI Series in Software Engineering, Addison-Wesley Professional (2015)
3. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: A Scalable Fully Distributed Web Crawler. Software: Practice and Experience 34(8), 711–726 (2004)
4. Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., Leymann, F.: Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains. International Journal On Advances in Software 7(3&4), 710–726 (2014)
5. Farenhorst, R., de Boer, R.C.: Knowledge management in software architecture: State of the art. In: Software Architecture Knowledge Management, pp. 21–38. Springer (2009)
6. Fehling, C., Barzen, J., Falkenthal, M., Leymann, F.: PatternPedia - Collaborative Pattern Identification and Authoring. In: PURPLSOC: Pursuit of Pattern Languages for Societal Change. pp. 252–284. epubli GmbH (2015)
7. Heydon, A., Najork, M.: Mercator: A Scalable, Extensible Web Crawler. World Wide Web 2(4), 219–229 (1999)
8. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
9. Hüttermann, M.: DevOps for Developers. Apress (2012)
10. Studer, R., Benjamins, V.R., Fensel, D.: Knowledge engineering: principles and methods. Data & knowledge engineering 25(1), 161–197 (1998)
11. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Babar, M.A.: A comparative study of architecture knowledge management tools. Journal of Systems and Software 83(3), 352–370 (2010)
12. Trinkle, P.: An Introduction to Unsupervised Document Classification (2009)
13. Wettinger, J., Andrikopoulos, V., Leymann, F.: Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications. In: Proceedings of the International Conference on Cloud Engineering (IC2E). IEEE (2015)
14. Wettinger, J., Breitenbücher, U., Kopp, O., Leymann, F.: Streamlining DevOps Automation for Cloud Applications using TOSCA as Standardized Metamodel. Future Generation Computer Systems 56, 317–332 (2016)