5 Combining API Documentation and Dependency Graph

5.1 Purpose of this Approach

In this approach, a website is built to combine API documentation of single services with a dependency graph, giving an overview of the whole system while still being able to focus on single services. The graph provides an overview of the system that is easy to understand and to remember and gives developers an idea of the communication patterns and the division of labor among the services. Through links on each node of the graph, the API documentation of the single service can be viewed on its own.

A combined API documentation containing all service endpoints in the system is also included and can be easily found on a node not connected to the graph and displayed on the top left side with the name of the system as a label. It allows developers who do not know which service a certain function belongs to scroll through all endpoints and use the search function of the browser to find keywords that might lead to it. Other advantages and a more detailed description of the combined API documentation can be found in chapter 4.

The graph can be generated by tools like Kiali or written by hand in the format defined in section 5.3.5). Other methods are possible too, as long as the resulting graphs can be converted to either the Kiali format or the format defined in this thesis, called standard format. It has to be uploaded to the website using the upload buttons and only one graph can be hold at a time. The graph is not guaranteed to be complete and the website does not check whether the dependencies given are correct.

Each node of the graph can hold a link to the API documentation of the corresponding service. The URLs for these links can be specified in the standard graph format. If the graph is uploaded in the Kiali format, it has to be downloaded by the user to add them. Nodes that are equipped with documentation are displayed in green, to make it easy to discern where documentation is missing. The nodes links can lead to any document. The website does not check for a certain format, to give each team the possibility to include any form of documentation they have. New services under development can be documented with tools that are used in the design phase or a

simple PDF listing endpoints or offering a short description, which is better than no documentation. This approach requires each team to provide an API documentation that is hosted on a server reachable through a URL.

5.2 Integration into Existing Tools

To generate the graph, the services are deployed in a Kubernetes cluster. Inside the cluster, Istio is intercepting all messages between the services and stores metadata about them. Kiali can use this data to generate a dependency graph. Since many microservice projects are already being managed by Kubernetes and Istio, this does not require many changes in the system. For projects that are managed differently, the graph can be generated by other tools or written by hand and converted to the standard graph format used by the website.

The JavaScript library dagre-d3 is used to render the graph in an SVG image, making it scalable and easy to attach to the DOM(Document Object Model) of the website. The library can be found here:
`https://github.com/dagrejs/dagre-d3`
The OpenAPI format is used to describe the APIs of the single services and the combined API of the system. Each API is displayed with SwaggerUI and hosted on a central documentation server or a server of the team working on the service. A link attached to each node leads to the API documentation of the corresponding service. Using the website with alternative API documentation methods is possible, as long as a link to any document is provided in the graph specification.

As discussed in chapter 4, swagger-combine and the novel tool sccp are used to combine all API specifications into a system API, which is connected to a special system node.

5.3 Usage

5.3.1 Quick Start

Use this command in the Website directory:

```
$ docker run -v $PWD/nginx/nginx.conf:/etc/nginx/nginx.conf -v $PWD:/var/www\
    -it -p 80:80 nginx
```

View the website in the browser on http://localhost and click the button `Upload Standard Graph`. Choose the graph specification located in
`Website/Graphs/graph_format.json`

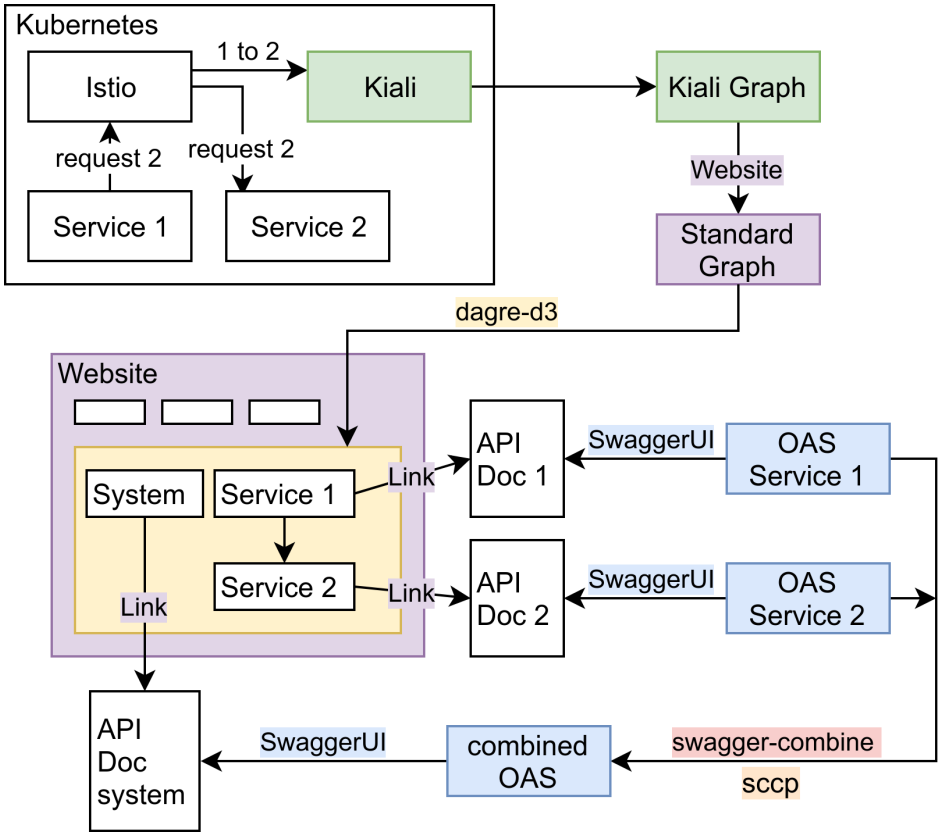

Figure 5.1: Integration of the novel Website into existing Documentation Tools

The website is now displaying a dependency graph of the BookInfo system with clickable nodes that lead to the API documentation of the corresponding services.

5.3.2 Generating Dependency Graph

To generate a dependency graph with Kiali, the system needs to be running in a Kubernetes cluster with Istio and Kiali installed and running as services inside the cluster. (Istio can also be used without Kubernetes, but this approach will not be discussed here). Examples of graph specifications can also be found on the CD in:

```
Website/Graphs
```

Instead of Kubernetes, minikube, a basic and simpler version of Kubernetes meant for small experiments, will be installed in this example. The commands are shown for the operating system Ubuntu.
Firstly, Docker and some other dependencies need to be installed with these commands:

```
$ sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent\
    software-properties-common socat
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64]\
    https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Minikube can then be installed and started:

```
$ sudo -i
$ curl -L\
   https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64\
   > /usr/bin/minikube && chmod +x /usr/bin/minikube
$ minikube start --vm-driver=none
$ snap install kubectl --classic
```

Istio and Kiali are installed with these commands:
(A demo is used here to omit the configuration steps needed to install Istio in a specific cluster, the demo also contains Kiali)

```
$ curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.2.5 sh -
$ cd istio-1.2.5
$ export PATH=$PWD/bin:$PATH
$ for i in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl apply -f $i;\
    done
$ kubectl apply -f install/kubernetes/istio-demo.yaml
```

BookInfo is a microservice system developed by Istio as an example. It is used here to demonstrate how to use the data collected by Istio to generate a graph in Kiali. To collect data on network communication in the BookInfo system, Istios sidecars need to be injected into all services and the system needs to be running to generate traffic:

```
# enable sidecar injection
$ kubectl label namespace default istio-injection=enabled
# start bookinfo application
$ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
# enable traffic from outside the cluster to reach bookinfo
$ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
# apply traffic rules
$ kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml
```

The data collected by Istio is automatically forwarded to Kiali. To generate traffic, requests need to be send to the BookInfo application:

```
# store address of kubernetes cluster
$ export INGRESS_PORT=$(kubectl -n istio-system get service\
    istio-ingressgateway -o\
    jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
$ export INGRESS_HOST=$(minikube ip)
$ export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
# send requests to bookinfo (request a list of all book titles)
$ curl -s http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
```

Kialis port needs to be forwarded to the standard port 80 so that the host machine knows that requests to its standard port should be forwarded to Kiali:

```
$ kubectl -n istio-system port-forward --address=116.203.10.109 $(kubectl -n\
    istio-system get pod -l app=kiali -o jsonpath='{.items[0].metadata.name}')\
    80:20001
```

The graph representation of the collected traffic metadata can be viewed in a browser (URL of the server that minikube is running on), as shown in figure 5.2. The graph

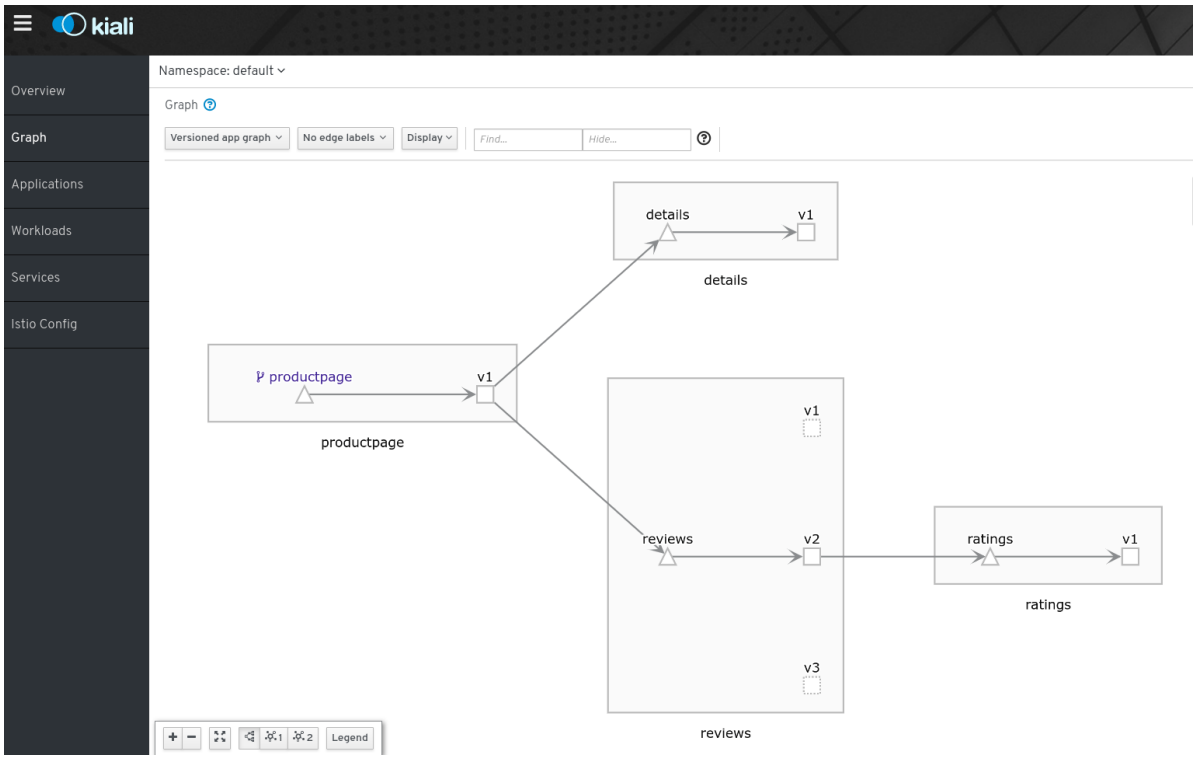

Figure 5.2: The kiali graph can be viewed and downloaded from a website provided by the kiali service

cannot be downloaded from the website, but it can be requested on Kialis API by typing this URL into a browser or curl command:

http://<server>/kiali/api/namespaces/graph?namespaces=default

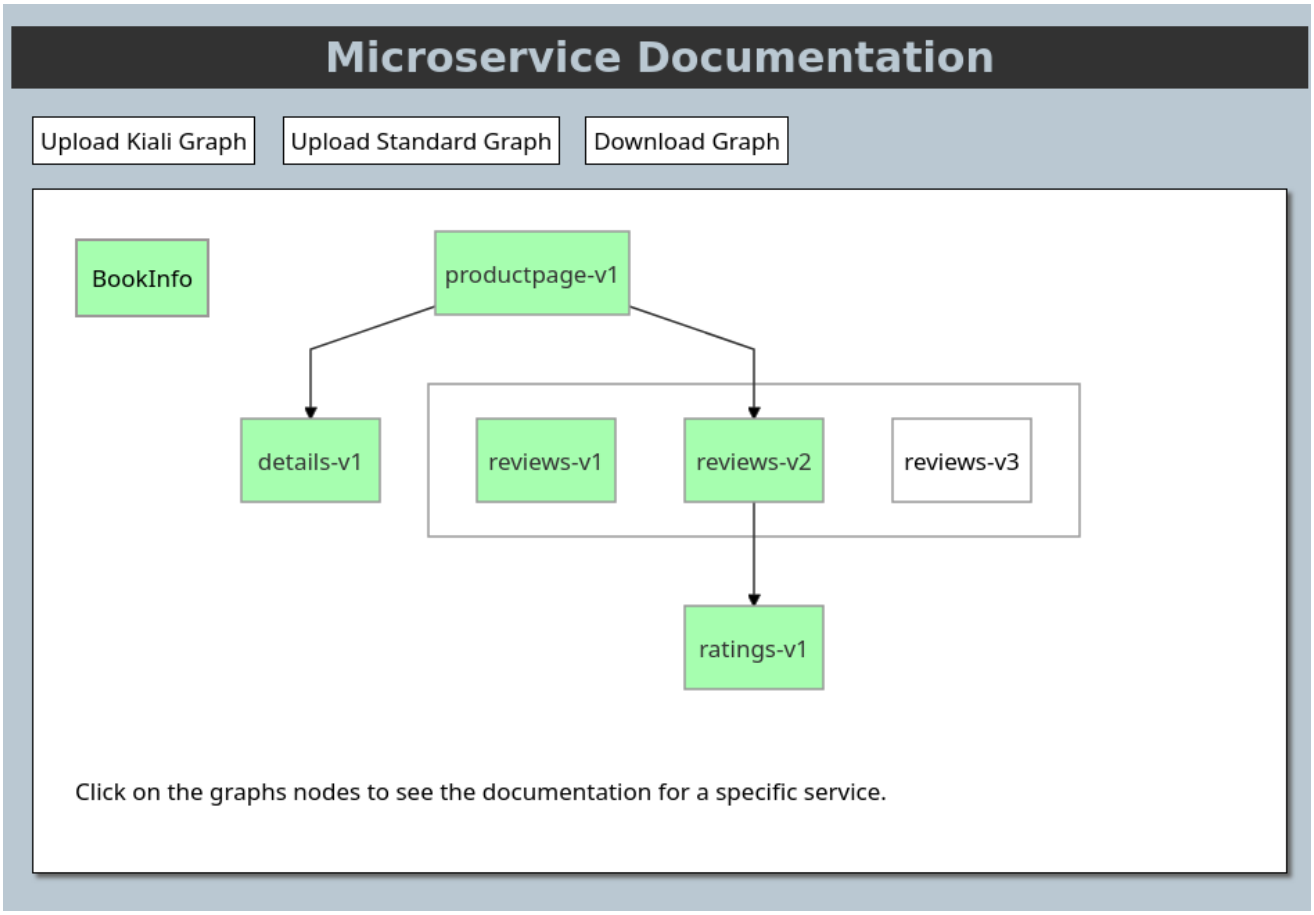

Figure 5.3: Graph as depicted by the novel Website. Nodes containing links to API Documentation are colored green

5.3.3 Hosting Website

The website can be found in the following directory on the CD:

Website/

Hosting the website can be done with a single command, given that Docker is installed on the host machine:

```
$ docker run -v $PWD/nginx/nginx.conf:/etc/nginx/nginx.conf -v $PWD:/var/www\
    -it -p 80:80 nginx
```

Docker automatically connects to Docker Hub, where it can download the container `nginx` and run it on the host machine. This container has all the necessary libraries

installed and is configured to run as a server. More details can be found here:
`https://hub.docker.com/_/nginx`
The command above copies all the files inside the current directory to the www-directory inside the nginx container, making them reachable on

```
http://localhost
```

The `index.html` file is automatically displayed first as the homepage.

5.3.4 Uploading Graph

Kiali and standard graphs in JSON format can be uploaded over two buttons on the top of the website. For a definition of the standard graph, see section 5.4.1. The graph is automatically displayed in the center element, it should look like figure 5.3. Links to API documentation and the system node are only displayed for graphs uploaded in the standard format, but every Kiali graph is automatically translated into the standard format on upload and can be downloaded to add links and information on the system and reuploaded afterwards.

5.3.5 Adding missing Dependencies and API Documentation

There is no possibility yet to edit the graph directly on the website. Its specification can be downloaded with a button on top of the website. Whether it was uploaded in Kiali or standard format, the downloaded file will always be in standard format.
A standard graph specification of the bookinfo system is shown in the listing below.

Links to API documentation can be added in the node elements with the keyword link. Missing nodes can be added by adding a node element to the nodes array. They need an id, a name (called app) and a version. Documentation links are optional. Multiple nodes can have the same name, if they are a different version of the same service. Services who share an app name are grouped together in the graph's depiction and a box is drawn around them.

Missing edges can be added in the edges array. They need a source and a target, matching the ids of the nodes they are connecting.

System API documentation can be added by adding a system element to the graph specification and setting its link element to the URL of the documentation.

```
{
  ”nodes”: [
    {
      ”id”: ”47efcb6a38cec94b8f02e15c58ce44df”,
      ”app”: ”details”,
      ”version”: ”v1”,
      ”link” :
        ”http://localhost/swagger−ui/index.html” +
        ”?url=http://localhost/bookinfo_swagger/details.yaml”
    },
    {
      ”id”: ”c7c4074158d3c18e0c780126e50637db”,
      ”app”: ”productpage”,
      ”version”: ”v1”,
      ”link” :
        ”http://localhost/swagger−ui/index.html” +
        ”?url=http://localhost/bookinfo_swagger/productpage.yaml”
    }, ...
  ],
  ”edges”: [
    {
      ”source”: ”11b370489b738638fabddc4f4ce47ebd”,
      ”target”: ”b744c74ec7104950b36f9bf0b47a22fd”
    }, ...
  ],
  ”system” : {
    ”name” : ”BookInfo”,
    ”link” : ”http://localhost/swagger−ui/index.html” +
      ”?url=http://localhost/bookinfo_swagger/combinedOAS.json”
  }
}
```

In this example, there is only one instance of the SwaggerUI website hosted on the same machine as the novel website. The API specifications are also hosted there and included into the SwaggerUI URLs as query parameters to point SwaggerUI to the API it should display.

5.4 Implementation of Website

The website is divided into three files: `index.html`, `styles.css` and `graph.js`. `index.html` is the first file read by the browser and defines the elements of the website, like titles, boxes and buttons. `styles.css` is included into index.html and further defines the style of the elements, like colors, size and shadows. The elements style can change over time, for example they can be made invisible by giving them the same color as the background and can later be made visible again. Such manipulations can be done by JavaScript. It is a programming language that turns a website from a static document with links to an interface with functionality like buttons that change the appearance of the website, data forms that can be used to transfer information from the user to the website and so on. To create such functionality, JavaScript functions are bound to the elements defined in the HTML file and executed when the user interacts with the element. In this case, the JavaScript code can be found in `graph.js`.

Three buttons are defined in the HTML file to upload and download graphs:

```
24 <input type="file" id="graph_file_kiali" accept=".json" class="filechooser"/>
25 <label for="graph_file_kiali">Upload Kiali Graph</label>
26 <input type="file" id="graph_file" accept=".json" class="filechooser"/>
27 <label for="graph_file">Upload Standard Graph</label>
28 <a id="download" class="button">Download Graph</a>
```

The upload buttons are defined as input elements that only accept JSON files and let the user choose a file from his own file system. After the user has chosen a file, the input elements report a change-event. These events are bound to the JavaScript functions `upload_kiali_graph()` and `upload_standard_graph()`:

```
224 document.addEventListener("DOMContentLoaded", function() {
225     document.getElementById('graph_file_kiali').addEventListener('change',
          ↪ upload_kiali_graph, false);
226   });
```

The graph specification is read in, stored in the local storage of the user's browser to prevent it from disappearing when the website is reloaded, and, if it is uploaded in Kiali format, converted to the standard format.:

```
12 var file = evt.target.files[0];
13  var reader = new FileReader();
14 reader.onload = function() {
15     kiali_graph_string = reader.result;
16     window.localStorage.setItem('graph', kiali_graph_string);
17     var kiali_graph_obj = JSON.parse(kiali_graph_string);
18     graph_obj = convert_kiali_to_standard(kiali_graph_obj);
19     draw_graph(graph_obj);
```

```
20 }
21 reader.readAsText(file);
```

The event `evt` is the change event of the upload button, in other words the upload of the graph specification. A reader object is created to handle the file that is uploaded and is instructed to read the file as a string as soon as the upload process is finished.

Kiali graph specifications are converted to the standard format before rendering:

```
184 var nodes = kiali_graph_obj.elements.nodes;
185 for(var i = 0; i < nodes.length; i++){
186     node = nodes[i];
187     graph_obj.nodes.push({
188         id : node.data.id,
189         app : node.data.app,
190         version : node.data.version
191     });
192 }
193
194 var edges = kiali_graph_obj.elements.edges;
195 for(var i = 0; i < edges.length; i++){
196     edge = edges[i];
197     graph_obj.edges.push({
198         source : edge.data.source,
199         target : edge.data.target
200     });
201 }
```

After storing the graph specification in a JavaScript object, it is rendered in the `draw_graph` function. In the first step, a graph object `g` is created from the graph specification in a format that is readable by the graph rendering library dagre-d3:

```
110 varnodes = graph_obj.nodes;
111 var clusters = [];
112 var nodes_included = [];
113 for(var i = 0; i < nodes.length; i++){
114     node = nodes[i];
115
116     // set the API documentation link of the node
117     g.setNode(node.id, { labelType: "html", label: "<a href=" + node.link + ">" +
        ↪ node.app + "-" + node.version + "</a>", width: node.app.length*10+10,
        ↪ height: 40, style: "fill: #afa"});
118
119     // create a cluster if the node is another version of an already existing app
120     if(nodes_included.includes(node.app)){
```

```
121             g.setNode(node.app, {label: node.app, clusterLabelPos: ""});
122             clusters.push(node.app);
123         }
124 }
125
126 // add the nodes to their corresponding clusters
127 for(var i = 0; i < nodes.length; i++){
128         node = nodes[i];
129         if(clusters.includes(node.app)){
130             g.setParent(node.id, node.app);
131         }
132 }
133
134 // add edges
135 var edges = graph_obj.edges;
136 for(var i = 0; i < edges.length; i++){
137         edge = edges[i];
138         g.setEdge(edge.source, edge.target);
139 }
140
141 dagre.layout(g);
```

The size of nodes is adjusted to the length of their label (the name of the service). Nodes that include a link to API documentation are colored green. Multiple versions of the same service are drawn as separate nodes, but are grouped in a cluster. To realize this, nodes are stored in an array and for every new node, the array is searched for another node with the same app name. If another node with the same name is found, a cluster is created as a node in the graph object `g` with a different style and the app name as name. After all nodes have been added to the graph object, the cluster node is set as a parent of all nodes with the same app name. Dagre-d3 draws the child nodes inside the parent node.

In the second step, the created graph object `g` is rendered by the dagre-d3 library, attached to the `svg` element on the website and centered:

```
157 var render = new dagreD3.render();
158 // attach graph to website element
159 var svg = d3.select("svg");
160 var svgGroup = svg.append("g");
161 // draw graph
162 render(d3.select("svg g"), g);
163 // center graph
164 document.getElementById("image").style.marginLeft = "" + -(g.graph().width / 2) +
    ↪ "px";
```

After a graph specification is uploaded, it is bound to the download button:

```
29 var button = document.getElementById(”download”);
30 var file = new Blob([JSON.stringify(graph_obj, undefined, 2)], {type: ’text/plain’});
31 button.href = URL.createObjectURL(file);
32 button.download = ’kiali_standard_graph.json’;
```

5.4.1 Graph Format Definition

```
Graph {
    nodes : [
        node{
            id : string
            app : string
            version : string
            link : string (URL)
        }
    ],
    edges : [
        edge{
            source : string (node id)
            target : string (node id)
        }
    ],
    system : {
        name : string
        link : string (URL)
    }
}
```

5.5 Fulfillment of Requirements

Requirements fulfilled by SwaggerUI for single services are also fulfilled in this approach, because SwaggerUI is used to document the single services. A detailed discussion on this topic can be found in section 3.3.1.
The system documentation is searchable, if the combined system API documentation is used to search for single endpoints, and it is scannable since the graph lists all services in a clear overview. An overview of the whole system is achieved by showing the whole systems interactions in the dependency graph and listing all endpoints in one document in the combined API documentation.

Focusing on one service at a time is possible by following the link to the single service API documentation. Multiple levels of detail are available in the SwaggerUI documentation and this approach adds a level with the dependency graph, where no details other than the names of the services and their dependence on each other is shown. Services relevant to a certain task can be easily identified by using the combined API documentation to find out which service is implementing a specific function, using the single service documentation to single out the relevant endpoints and use the dependency graph to find out which other services could be executing a subtask of the relevant function. Naming the services and endpoints after the function they fulfill in the system and including a description that includes all the relevant keywords is required

Each team has the freedom to use their favored documentation tools, since the links can lead to any document, making this approach flexible. However, if the teams use different documentation methods, the combined API documentation is not guaranteed to contain all services and the swagger-combine approach is only possible with documents in the OpenAPI format. If the combined API documentation is needed, all teams should agree on one documentation format, but it could be a different one than OpenAPI. Graphs can be generated by Kiali or any other tool, as long as it is possible to convert the specification to either the standard format defined here or the Kiali format. The website could also be altered to display other graphical system documentation, since the dagre-d3 library is able to draw all kinds of graphs.

Both the graph specification and the API specifications are machine-readable and stored in the JSON format, for which libraries exist for most programming languages. The solution can be integrated in the development process by including the OpenAPI specification in code annotations and using a script to automatically update the specification when a new version of code is uploaded to a repository. The combination of these specifications and the upload to their designated URLs could also be included in the script. If the system is running in a kuernetes cluster, a new graph could be generated every day and automatically replace the old graph. But a review of the graph by a human would still be necessary, since Kiali can not guarantee completeness and the API documentation links need to be set. The process of merging new graph data with the old graph while keeping the links intact could be the topic of future work.

- ✓ Separate Documentation for each Service
- ✓ Overview of Whole System
- ✓ Searchable
- ✓ Scannable
- ✓ Multiple Levels of Detail
- ✓ Quick Identification of Relevant Services
- ✓ Easy to Update
- ✓ Time-Saving for Developers
- ✓ Integration into Development Process
- ✓ Flexibility
- ✓ Accessible Platform
- ✓ Machine Readability