(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
04 April 2019 (04.04.2019)

WIPO | PCT

(10) International Publication Number
WO 2019/068037 A1

(51) International Patent Classification:
G06F 11/36 (2006.01) *G06F 9/50* (2006.01)
G06F 8/60 (2018.01)

(21) International Application Number:
PCT/US2018/053628

(22) International Filing Date:
28 September 2018 (28.09.2018)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/566,351 30 September 2017 (30.09.2017) US

(71) Applicant: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway M/S 5OP7, Redwood Shores, California 94065 (US).

(72) Inventors: **CALDATO, Claudio**; 21926 NE 20th Way, Sammamish, Washington 98074 (US). **SCHOLL, Boris**; 8530 NE 128th Street, Kirkland, Washington 98034 (US).

(74) Agent: **BERGSTROM, James T.** et al.; 1100 Peachtree Street NE, Suite 2800, Mailstop: IP Docketing - 22, Atlanta, Georgia 30309 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

WO 2019/068037 A1

(54) Title: REAL-TIME DEBUGGING INSTANCES IN A DEPLOYED CONTAINER PLATFORM

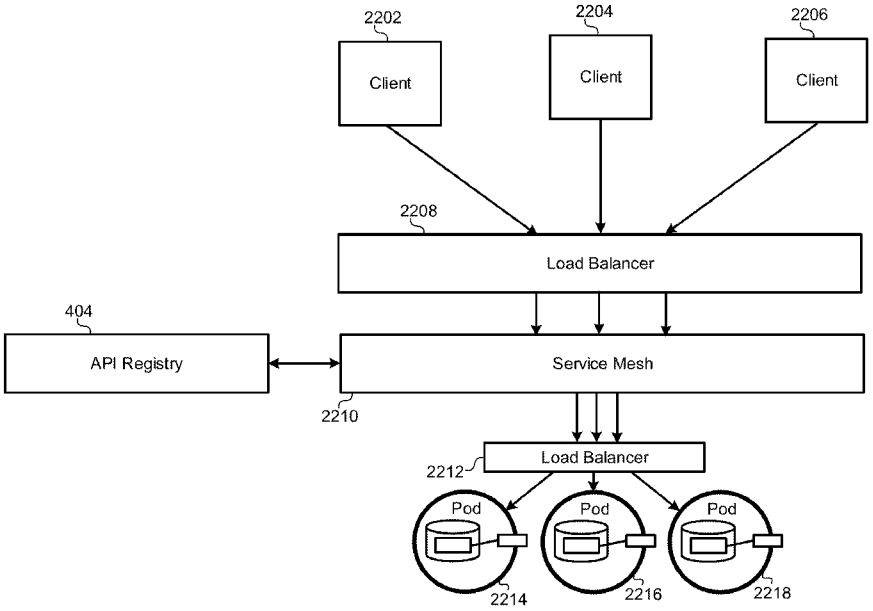

FIG. 22

(57) Abstract: A method may include receiving a request for a service at a container environment. The container environment may include a service mesh and a plurality of services encapsulated in a plurality of containers. The service may be encapsulated in first one or more containers. The method may also include determining that the request should be routed to a debug instance of the service; and instantiating the debug instance of the service. The debug instance may be encapsulated in second one or more containers and may include code implementing the service and one or more debugging utilities. The method may additionally include routing, by the service mesh, the request to the debug instance.

[Continued on next page]

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

REAL-TIME DEBUGGING INSTANCES IN A DEPLOYED CONTAINER PLATFORM

CROSS-REFERENCES TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Application No. 62/566,351 filed on September 30, 2017, which is incorporated herein by reference. This application is also related to the following commonly assigned applications filed on the same day as this application, each of which is also incorporated herein by reference:

- U.S. Patent Application No. __/___,___ filed on September __, 2018, titled API REGISTRY IN A CONTAINER PLATFORM FOR AUTOMATICALLY GENERATING CLIENT CODE LIBRARIES (Attorney Docket No. 088325-1090745);
- U.S. Patent Application No. __/___,___ filed on September __, 2018, titled API REGISTRY IN A CONTAINER PLATFORM PROVIDING PROPERTY-BASED API FUNCTIONALITY (Attorney Docket No. 088325-1090746);
- U.S. Patent Application No. __/___,___ filed on September __, 2018, titled DYNAMIC NODE REBALANCING BETWEEN CONTAINER PLATFORMS (Attorney Docket No. 088325-1090747);
- U.S. Patent Application No. __/___,___ filed on September __, 2018, titled OPTIMIZING REDEPLOYMENT OF FUNCTIONS AND SERVICES ACROSS MULTIPLE CONTAINER PLATFORMS AND INSTALLATIONS (Attorney Docket No. 088325-1090748);

BACKGROUND

[0002] In the abstract, containers in any form represent a standardized method of packaging and interacting with information. Containers can be isolated from each other and used in parallel without any risk of cross-contamination. In the modern software world, the term "container" has gained a specific meaning. A software container, such as a Docker® container, is a software construct the logically encapsulates and defines a piece of software. The most common type of software to be encapsulated in the container is an application, service, or microservice. Modern

containers also include all of the software support required for the application/service to operate, such as an operating system, libraries, storage volumes, configuration files, application binaries, and other parts of a technology stack that would be found in a typical computing environment. This container environment can then be used to create multiple containers that each run their own services in any environment. Containers can be deployed in a production data center, an on-premises data center, a cloud computing platform, and so forth without any changes. Spinning up a container on the cloud is the same as spinning up a container on a local workstation.

[0003] Modern service-oriented architectures and cloud computing platforms break up large tasks into many small, specific tasks. Containers can be instantiated to focus on individual specific tasks, and multiple containers can then work in concert to implement sophisticated applications. This may be referred to as a microservice architecture, and each container can use different versions of programming languages and libraries that can be upgraded independently. The isolated nature of the processing within containers allows them to be upgraded and replaced with little effort or risk compared to changes that will be made to a larger, more monolithic architectures. Container platforms are much more efficient than traditional virtual machines in running this microservice architecture, although virtual machines can be used to run a container platform.

BRIEF SUMMARY

[0004] In some embodiments, a method of providing runtime debugging for containerized services in container environments may include receiving a request for a service at a container environment. The container environment may include a service mesh and a plurality of services encapsulated in a plurality of containers. The service may be encapsulated in first one or more containers. The method may also include determining that the request should be routed to a debug instance of the service; and instantiating the debug instance of the service. The debug instance may be encapsulated in second one or more containers and may include code implementing the service and one or more debugging utilities. The method may additionally include routing, by the service mesh, the request to the debug instance.

[0005] In some embodiments, a non-transitory, computer-readable medium may include instructions that, when executed by one or more processors, causes the one or more processors to

perform operations including receiving a request for a service at a container environment. The container environment may include a service mesh and a plurality of services encapsulated in a plurality of containers. The service may be encapsulated in first one or more containers. The operations may also include determining that the request should be routed to a debug instance of the service; and instantiating the debug instance of the service. The debug instance may be encapsulated in second one or more containers and may include code implementing the service and one or more debugging utilities. The operations may additionally include routing, by the service mesh, the request to the debug instance.

[0006] In some embodiments, a system may include one or more processors and one or more memory devices comprising instructions that, when executed by the one or more processors, cause the one or more processors to perform operations including receiving a request for a service at a container environment. The container environment may include a service mesh and a plurality of services encapsulated in a plurality of containers. The service may be encapsulated in first one or more containers. The operations may also include determining that the request should be routed to a debug instance of the service; and instantiating the debug instance of the service. The debug instance may be encapsulated in second one or more containers and may include code implementing the service and one or more debugging utilities. The operations may additionally include routing, by the service mesh, the request to the debug instance.

[0007] In any embodiments, any or all of the following features may be included in any combination and without limitation. The first one or more containers may be organized into a container pod. The container environment may include an orchestrated container platform comprising a container scheduler. The container scheduler may cause the debug instance of the service to be instantiated. The container environment may include an Application Programming Interface (API) registry that causes the debug instance of the service to be instantiated. The API registry may receive a registration for the debug instance of the service and makes an HTTP endpoint of the debug instance of the service available through an API function call. The API registry may receive a registration for the service comprising a property indicating that the debug instance of the service should be instantiated. The service may be encapsulated in a single container. The single container may also include the one or more debugging utilities. The one or more debugging utilities may be encapsulated in at least one container other than the single container. The one or more debugging utilities may include a process for monitoring memory

usage or processor usage. The one or more debugging utilities may include a debug daemon. The code implementing the service may include a debug build of the service. The debug instance of the service may be instantiated prior to receiving the request. The debug instance of the service may be instantiated in response to receiving the request. Determining that the request should be routed to the debug instance of the service may include identifying a source of the request. Determining that the request should be routed to the debug instance of the service may include recognizing a header in the request that designates the request as a debug request. The request may be forwarded to the debug instance of the service without interrupting the routing of other requests to the service.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] A further understanding of the nature and advantages of the present invention may be realized by reference to the remaining portions of the specification and the drawings, wherein like reference numerals are used throughout the several drawings to refer to similar components. In some instances, a sub-label is associated with a reference numeral to denote one of multiple similar components. When reference is made to a reference numeral without specification to an existing sub-label, it is intended to refer to all such multiple similar components.

[0009] **FIG. 1** illustrates a software structure and logical arrangement of development and runtime environments for services in a container platform, according to some embodiments.

[0010] **FIG. 2** illustrates a specialized computer hardware system that is specifically designed to run the embodiments described herein.

[0011] **FIG. 3** illustrates a data organization that may be specific to the container platform used by some of the embodiments described herein.

[0012] **FIG. 4** illustrates an API registry that can be deployed to the IDE and the production/runtime environment, according to some embodiments.

[0013] **FIG. 5** illustrates the deployment of the API registry for use with the container platform at runtime, according to some embodiments.

[0014] **FIG. 6A** illustrates a flowchart of a method for deploying the API registry, according to some embodiments.

[0015] **FIG. 6B** illustrates a software structure of a container platform when the API registry is deployed using the flowchart in FIG. 6A, according to some embodiments.

[0016] **FIG. 7A** illustrates a flowchart of a method for registering a service with the API registry, according to some embodiments.

[0017] **FIG. 7B** illustrates a hardware/software diagram of the steps for registering an API with the API registry, according to some embodiments.

[0018] **FIG. 8** illustrates examples of a graphical interface and a command line interface for browsing and selecting APIs that are registered with the API registry, according to some embodiments.

[0019] **FIG. 9** illustrates a flowchart of a method for using a service and its corresponding function registered with the API registry, according to some embodiments.

[0020] **FIG. 10** illustrates how a selection may be received by the API registry through the graphical interface of the CreateUser( ) function.

[0021] **FIG. 11** illustrates an example of a client library generated automatically for a service by the API registry, according to some embodiments.

[0022] **FIG. 12** illustrates an embodiment of a client library that accommodates dynamic binding between service endpoints and API functions, according to some embodiments.

[0023] **FIG. 13** illustrates an embodiment of a client library that can marshal additional data to complete an input data set for a service call, according to some embodiments.

[0024] **FIG. 14** illustrates a client library that can handle retries when calling a service, according to some embodiments.

[0025] **FIG. 15A** illustrates a method of providing API properties to the API registry, according to some embodiments.

[0026] **FIG. 15B** illustrates a hardware/software diagram of how a service can provide API properties to the API registry, according to some embodiments.

[0027] **FIG. 16** illustrates a hardware/software diagram where a property is used by the API registry to deploy a service with high availability, according to some embodiments.

[0028] **FIG. 17** illustrates a hardware/software diagram of a property that enforces end-to-end encryption through the API registry, according to some embodiments.

[0029] **FIG. 18** illustrates a property for an API registry to implement usage logging for a service 1808, according to some embodiments.

[0030] **FIG. 19** illustrates a hardware/software diagram of a property that can enforce an authentication protocol for a service, according to some embodiments.

[0031] **FIG. 20** illustrates a hardware/software diagram for a property that enables runtime instantiation of a service, according to some embodiments.

[0032] **FIG. 21** illustrates a hardware/software diagram of a property that implements a rate limiting function for a service, according to some embodiments.

[0033] **FIG. 22** illustrates a block diagram of a portion of the cloud computing platform for receiving service requests, according to some embodiments.

[0034] **FIG. 23** illustrates a debug build of a service encapsulated in a pod, according to some embodiments.

[0035] **FIG. 24** illustrates an alternative pod for a debug build of a service, according to some embodiments.

[0036] **FIG. 25** illustrates a block diagram of a system for instantiating a debug instance of a service, according to some embodiments.

[0037] **FIG. 26** illustrates a block diagram of the container platform routing debug requests to the debug instance of the service, according to some embodiments.

[0038] **FIG. 27** illustrates a block diagram of a cloud computing platform that clones requests, according to some embodiments.

[0039] **FIG. 28** illustrates a block diagram of cloned requests being forwarded to a debug instance of the service, according to some embodiments.

[0040] **FIG. 29** illustrates a flowchart of a method for providing runtime debugging for containerized services in container environments.

[0041] **FIG. 30** illustrates a simplified block diagram of a distributed system for implementing some of the embodiments.

[0042] **FIG. 31** illustrates a simplified block diagram of components of a system environment by which services provided by the components of an embodiment system may be offered as cloud services.

[0043] **FIG. 32** illustrates an exemplary computer system, in which various embodiments may be implemented.

DETAILED DESCRIPTION

[0044] Described herein, are embodiments for an Application Programming Interface (API) registry that is part of an Integrated Development Environment (IDE) that allows developers to register services during development and make those services available to other services both during and after deployment. The API registry can be deployed as part of an orchestrated container platform, operating as a containerized application on the container platform. As services or microservices are developed and deployed into containers on the container platform, the API registry can execute a discovery process to locate available endpoints (e.g., IP addresses and port numbers) within the container platform that correspond to available services. The API registry can also accept an upload of an API definition file that can be used to turn the raw service endpoint into an API function made available through the API registry. The API registry can dynamically bind the discovered endpoint to an API function that be kept up-to-date and made available to other services in the container platform. This provides a stable endpoint that other services can statically call while the API registry manages any changes to the binding between the API function in the service endpoint. This also simplifies the process for using services in the container platform. Instead of writing code for an HTTP call, new services can simply use the API interface to access registered services.

[0045] In some embodiments, the IDE can provide a navigation/browse interface for developers to locate services that are available in the container platform and registered with the

API registry. When calls to existing services are created by the API registry for new services under development, the API registry can automatically generate a set of client libraries that include all the necessary functionality to interact with the registered service. For example, some embodiments may generate an object class that includes member functions corresponding to API calls. During development, new services can simply instantiate these objects and/or use their member functions to make a call to the corresponding API. The code in the client libraries governs a direct connection between the calling service and the endpoint of the registered service and may include code that handles all the functionality necessary for this interaction. For example, the automatically generated client libraries may include: code for packaging and formatting parameters from the API call into an HTTP call to the service endpoint, code for marshaling data to complete parameter sets for the call, code for packaging information into a compatible packet (JSON, XML, etc.), code for receiving and parsing result packets, code for handling retries and error conditions, and so forth. From the calling service's perspective, the code to handle all of this functionality is automatically generated by the API registry and therefore abstracts and encapsulates the details of the service call into the client library object. All that is required of the calling service is to execute a member function of the client library object created by the API registry.

[0046] In some embodiments, the API registry can also accept an upload of a set of properties that may define the runtime execution of the registered service. This set of properties can be uploaded during development along with the API definition file. These properties can define runtime characteristics, such as end-to-end encryption, usage/logging requirements, user authentication, on-demand service instantiation, multiple service deployment instances for high availability, rate/usage limiting, and other runtime characteristics. The API registry can ensure that these properties are met by interacting with the container environment during development, during deployment, and during runtime. During development, the automatically generated client libraries for calling services can include code that may be required to execute these properties, such as encryption code, usage logging code, and/or interaction with a user authentication service. When a registered service is being deployed, the API registry can instruct the container platform to instantiate multiple instances of the service and/or additional load-balancing modules to ensure high reliability of the service during runtime. During runtime when a service is called,

the API registry can cause the service to be instantiated for on-demand instantiation, limit the number of API calls that can be made to throttle usage, and perform other runtime functions.

[0047] **FIG. 1** illustrates a software structure and logical arrangement of development and runtime environments for services in a container platform, according to some embodiments. The environments may include an IDE 102 that may be used to develop services and microservices to be deployed on a container platform. An IDE is a software suite that consolidates and provides all of the basic tools that service developers can use to write and test new services. The IDE 102 may include a source code editor 106 with a graphical user interface (GUI), code completion functions, and navigate/browse interfaces that allow a developer to write, navigate, integrate, and visualize the source-code-writing process. The IDE 102 may also include a debugger 110 that includes variable interfaces, immediate variable interfaces, expression evaluation interfaces, memory content interfaces, breakpoint visualization and functionality, and other debugging functions. The IDE 102 may also include a compiler and/or interpreter 108 for compiling and running compiled machine code or interpreted byte code. The compiler/interpreter 108 can include build tools that allow developers to use/generate makefiles another build automation constructs. Some embodiments of the IDE 102 may include code libraries 112 that include common code functions, objects, interfaces, and/or other structures that can be linked into a service under development and reused across multiple developments.

[0048] Services can be developed and thoroughly tested within the IDE 102 until they are ready for deployment. The services can then be deployed to a production/deployment environment 104. The production/development environment 104 may include many different hardware and/or software structures, including dedicated hardware, virtual machines, and containerized platforms. Prior to this disclosure, when a service 114 was deployed into the production/deployment environment 104, the service 114 would no longer have runtime access to many of the tools used in the IDE 102. Any functionality needed by the service 114 to run in the production/development environment 104 needed to be packaged from the code libraries 112 and deployed with the service 114 into the production/deployment environment 104. Additionally, the service 114 would typically be deployed without any of the functionality for the debugger 110 or a copy of the source code from the source code editor 106. Essentially, the service 114 would be deployed to the production/deployment environment 104 with all of the

functionality required for runtime operation, but would be stripped of the information that was only used during development.

[0049] **FIG. 2** illustrates a specialized computer hardware system that is specifically designed to run the embodiments described herein. By way of example, the service 114 can be deployed into an Infrastructure as a Service (IaaS) cloud computing environment 202. This is a form of cloud computing that provides virtualized or shared computing resources over a network. The IaaS cloud computing environment 202 may also include or be coupled with other cloud computing environments arranged as Software as a Service (SaaS) and/or Platform as a Service (PaaS) architetures. In this environment, the cloud provider can host an infrastructure of hardware and/or software components that were traditionally present in an on-premises data center. This hardware may include servers, storage, networking hardware, disk arrays, software libraries, and virtualization utilities such as a hypervisor layer. The IaaS environment 202 can be provided by a commercial source, such as Oracle® or other publicly available cloud platforms. The IaaS environment 202 may also be deployed as a private cloud using a private infrastructure of hardware and software.

[0050] Regardless of the type of cloud environment, the service 114 can be deployed onto a number of different types of hardware/software systems. For example, the service 114 can be deployed to dedicated hardware 206. The dedicated hardware 206 may include hardware resources, such as servers, disks, operating systems, software packages, and so forth, that are specifically assigned to the service 114. For example, a specific server may be allocated to handle traffic flowing to and from the service 114.

[0051] In another example, the service 114 can be deployed to hardware/software that is operated as one or more virtual machines 208. A virtual machine is an emulation of a computer system that provides the functionality of the dedicated computer hardware 206. However, instead of being dedicated to a specific function, the physical hardware can be shared by number of different virtual machines. Each virtual machine can provide all the functionality needed to execute including a complete operating system. This allows virtual machines having different operating systems to run on the same physical hardware and allows multiple services to share a single piece of hardware.

[0052] In a another example, the service 114 can be deployed to a container platform 210. The container platform differs from the virtual machines 208 in a number of important ways. First, the container platform 210 packages individual services into containers as described in greater detail below in FIG. 3. Each container shares a host operating system kernel, and they also share binaries, libraries, and other read-only components. This allows containers to be exceptionally light – often only a few megabytes in size. Additionally, a lightweight container is very efficient, taking just seconds to start versus the minutes required to boot up a virtual machine. Containers also reduce management overhead by sharing the operating system and other libraries that can be maintained together for the entire set of containers in the container platform 210. Even though containers share the same operating system, they provide an isolated platform, as the operating system provides virtual-memory support for isolation. Container technologies may include Docker® containers, the Linux Libcontainer®, the Open Container Initiative (OCI), Kubernetes®, CoeOS, Apache® Mesos, along with others. These containers can be deployed to a container orchestration platform, which may be referred to herein as simply the "container platform" 210. A container platform manages the automated arrangement, coordination, and management of deployed software containers. The container platform 210 can provide service discovery, load-balancing, health checks, multiple deployments, and so forth. The container platform 210 may be implemented by any publicly available container platform, such as Kubernetes, that runs containers organized in nodes and pods.

[0053] Regardless of the platform 206, 208, 210 on which the service 114 is deployed, each of the platforms 206, 208, 210 can provide service endpoints 212, 214, 216 that provide public access for calling the service 114. Generally, these endpoints can be accessed through an HTTP call and are associated with an IP address and a port number. By connecting to the correct IP address and port number, other services can call services deployed to any of the platforms 206, 208, 210 when they are made publicly available. Each service, such as service 114, may include its own proprietary formats and data requirements for calling the service. Similarly, each service may return results that are specific in format and data type to that service 114. In addition to the service-specific requirements, the particular deployment platform 206, 208, 210 may also include additional requirements for interacting with the service 114, such as programming languages, package formats (JSON, XML, etc.) that need to be complied with to properly interact with the service, and so forth.

[0054] Although the examples above allow the service 114 to be deployed to any of the described platforms 206, 208, 210, the embodiments described herein are specifically designed for the container platform 210 described above. Thus, embodiments that are specifically recited to be deployed in a "container platform" can be distinguished from other embodiments that are specifically recited to be deployed in a virtual machine platform, on the server or dedicated hardware platform, or generally in an IaaS environment.

[0055] **FIG. 3** illustrates a data organization that may be specific to the container platform 210 used by some of the embodiments described herein. Generally, any deployment of a service to the container platform will be deployed to a pod 304, 306. A pod is an abstraction that represents a group of one or more application containers (e.g., Docker or rkt). A pod may also include some shared resources that are commonly available to all of the containers within the pod. For example, pod 304 includes container 310 and container 312. Pod 304 also includes a shared resource 308. The resource may include a storage volume or other information about how containers are run or connected within the pod 304. The pod 304 can model an application-specific logical host that contains different service containers 310, 312 that are relatively tightly coupled. For example, service 326 in container 310 can utilize the resource 308 and call service 320 in container 312. Service 320 can also call service 322, which in turn calls service 324, each of which are deployed to container 312. The output of service 324 can be provided to a network IP address and port 318, which is another common resource shared by the pod 304. Thus, the services 320, 322, 324, 326 all work together with the shared resource 308 to provide a single service that can be accessed by the IP address and port numbe 318 by services run in other containers. The service can also be accessed through the IP address and port 318 by computer systems that are external to the container platform, such as a workstation, a laptop computer, a smart phone, or other computing device that is not part of the container platform or IaaS environment.

[0056] In the simplest deployment, each container may include a single service, and each pod may include a single container that encapsulates the service. For example, pod 306 includes only a single container 314 with a single service 328. The single service is accessible through the IP address and port number 316 of the pod 306. Typically, when a service is deployed to the container platform, a container and a pod will be instantiated to hold the service. A number of different pods can be deployed to a container node 302. Generally, pods run within nodes. A

node represents a worker machine (either virtual or physical) in the container platform. Each node is managed by a "master" that automatically handles scheduling pods within each of the nodes. Each node can run a process that is responsible for communication between the master and the node and for managing the pods in containers on the machine represented by the node. Each node may also include a container runtime responsible for pulling a container image from a registry, unpacking the container, and running the service.

[0057] **FIG. 4** illustrates an API registry 404 that can be deployed to the IDE 102 and the production/runtime environment 104, according to some embodiments. As described above, a technical problem exists wherein when the service 114 is deployed from the IDE 102 to the production/deployment environment 104, the service 114 loses runtime access to information that is exclusively available in the IDE 102. The API registry 404 is accessible by the service 114 while it is deployed and operating during runtime in the production/development environment 104. The previous technical problem that isolated development functions from runtime functions is overcome by the API registry 404 by the registration of services with the API registry 404 during development and providing an API definition and/or API properties to the API registry 404. The information defining the API can be used by new services in development in the IDE 102 as well as services that have been deployed to the production/deployment environment 104. After this registration process is complete, the service 114 can operate using client libraries that access the API registry 404 during runtime to ensure that the API functions are correctly bound to the current IP address and port number of the corresponding service. The API registry 404 represents a new data structure and processing unit that was specifically designed to solve these technical problems.

[0058] Another technical problem that existed in the art was implementing service properties as they are deployed to the production/development environment 104. For example, if a service was to be deployed with high availability, the developer would need to build container deployment files that specifically instantiated multiple instances of the service in the container platform and balanced traffic in such a way that the service was always available. Service developers did not always have this expertise, nor were they often able to manage the deployment of their service. As described below, the API registry 404 allows a service to simply select properties, such as high availability, that can then be implemented automatically by the

API registry 404. This technical solution is possible because the API registry 404 bridges the gap between the IDE 102 and the production/deployment environment 104.

[0059] **FIG. 5** illustrates the deployment of the API registry 404 for use with the container platform 210 at runtime, according to some embodiments. One of the technical solutions and improvements to the existing technology offered by the API registry 404 is the maintenance of stable endpoints for service calls, as well as the simplification and automatic code generation for accessing the service calls. Prior to this disclosure, calls between services were point-to-point connections using, for example, an HTTP call to an IP address and port number. As services are updated, replaced, relocated, and redeployed in the container platform 210, the IP address and port number may change frequently. This required all services that called an updated service to update their IP address and port numbers in the actual code that called that service. The API registry 404 solves this technical problem by providing a dynamic binding between the IP address and port number of a service and an API function that is made available through the API registry. The client libraries that are automatically generated by the API registry 404 can include a function that accesses the API registry 404 to retrieve and/or verify a current IP address and port number for a particular service. Thus, a first service connecting to a second service need only perform a one-time generation of a client library to provide a lifetime-stable connection to the second service.

[0060] Another technical problem solved by the API registry 404 is the automatic generation of client libraries. Prior to this disclosure, a first service accessing a second service required the developer to write custom code for accessing the second service. Because this code could change over time, incompatibilities would a rise between the first and second services that required updates to both services. The API registry 404 solves this technical problem by uploading an API definition file that is used to automatically generate client libraries for calling services. Therefore, a service can specify specifically how the calling code in any other service should operate, which guarantees compatibility. These client libraries also greatly simplify and encapsulate the code for calling the service. As described below, a complicated HTTP call using IP address and a port numbers can be replaced with a simple member function call in a language that is specific to the calling service (e.g., Java, C#, etc.). This allows a calling service to select an API function from the API registry 404, and the code that implements at function can be downloaded to the calling service as a client library.

[0061] **FIG. 6A** illustrates a flowchart of a method for deploying the API registry 404, according to some embodiments. The method may include deploying the API registry service to the container environment (601). The API registry can be implemented as a service operating in the container environment within the container. Thus, the API registry can be actively running after services are deployed within the container environment such that it can be accessed at run time. The API registry can also be linked to the existing IDE described above. The method may further include discovering ports for available services in the container platform (603). As services are deployed to the container platform, the API registry can launch a discovery process that sequentially traverses each of the services deployed to the container platform. For each service, the API registry can detect and record an IP address and a port number. The listing of IP address and port numbers discovered by this process can be stored in a data structure, such as a table associated with the API registry. Each IP address and port number can also be stored with a name for the service or other identifier that uniquely identifies the service on the container platform. These initial steps shown in flowchart in FIG. 6A provide a starting point for the API registry to begin operating in the runtime environment of the container platform and to be available to services under development in the IDE.

[0062] **FIG. 6B** illustrates a software structure of the container platform 210 when the API registry is deployed using the flowchart in FIG. 6A, according to some embodiments. As described above, the API registry 404 can be deployed to a container 620 in the container platform 210. The container 620 can operate within one or more pods and within a node as described above in FIG. 3. The API registry 404 can be made privately available to any of the other containers in the container platform 210. In some embodiments, the API registry 404 can also be made publicly available to other devices that are not part of the container platform 210. As a containerized service, the API registry 404 may have an IP address and port number that are available to other services. However, the IP address and port number of the API registry 404 would only be used by the code that is automatically generated in client libraries, therefore some embodiments do not need to publish the IP address and port number for the API registry 404. Instead, the client libraries in the IDE itself can maintain an up-to-date listing of the IP address and port number for the API registry 404 such that it can be contacted during development, deployment, and runtime of other services.

[0063] After deploying the API registry 404 to the container 620, the API registry 404 can execute a discovery process. The discovery process can use a directory listing for nodes in the container platform to identify pods that implement services with an IP address and port number. The API registry 404 can then access a unique identifier, such as a number or name for each available service, and store an identifier with each IP address and port number in the container platform 210. This discovery process can be periodically executed to detect new services that are added to the container platform 210, as well as to identify existing services that are removed from the container platform 210. As described below, this discovery process can also be used to detect when an IP address and port number change for an existing service. For example, the API registry 404 can discover services having endpoints 602, 604, 606, 608. In the process described below, the API registry 404 can bind each of these endpoints 602, 604, 606, 608 to an API function that is registered with the API registry 404. At some point after this initial discovery, the IP address and/or port number for endpoint 602 may be changed when the service associated with endpoint 602 is replaced, updated, or revised. The API registry 404 can detect this change to endpoint 602 and update a binding to an existing API function provided by the API registry 44.

[0064] Similarly, the API registry 404 can use the discovery process to detect when endpoints are no longer available, and then remove the API functions associated with the service. In some embodiments, when a service has been registered with the API registry 404, but the corresponding API functions are not currently bound to a valid endpoint, the API registry 404 can provide a mock response to any service calling the corresponding API functions. For example, if an API has been registered for the service corresponding to endpoint 604, but endpoint 604 is not currently available, the API registry 404 can intercept a call made to endpoint 604 and provide default or dummy data in response. This allows services that call the service associated with endpoint 604 to maintain functionality and/or continue the design process without "breaking" the connection to this particular service. Mock/testing data scenarios will be described in greater detail below.

[0065] **FIG. 7A** illustrates a flowchart of a method for registering a service with the API registry 404, according to some embodiments. The method may include receiving an upload of an API definition (701). The API definition may be provided in the form of a data packet, file, or a link to an information repository. The API definition may include any information that can

be used to identify and define API functions that should be bound to endpoints associated with the service. For example, some embodiments of the API definition may include the following data: a service name or other unique identifier; function names corresponding to service endpoints and calls, data inputs required to call the service with corresponding descriptions and data types; result data formats and data types; a current IP address and/or port number; documentation that describes the functionality of the API functions that will be associated with the endpoint; default or dummy data values that should be returned during mock/test scenarios; and any other information that may be used by the API registry 404 to translate the HTTP request received by the endpoint into a client library that uses API function calls of class data objects.

[0066] The method may also include creating corresponding API functions based on the uploaded API definitions (703). These API functions can be generated automatically based on the API definition. Each endpoint for a service may be associated with a plurality of different API functions. For example, an endpoint implementing a RESTful interface may receive HTTP calls for POST, GET, PUT, and DELETE functions at the same IP address and port number. This may result in, for example, for different API functions. For example, if the interface represents a list of users, this can correspond to at least four different API functions, such as GetUser( ), AddUser( ), RemoveUser( ), and UpdateUser( ). Additionally, each API function may include a number of different parameter lists, such as UpdateUser(id), UpdateUser(name), UpdateUser(firstname, lastname), and so forth. These API functions can be generated and made available to other services through the API registry. As will be described in greater detail below, it should be noted that services are not required to call these functions through the API registry. Instead, these functions are made available to browse in the API registry, and when selected, the API registry can generate client libraries that implement these functions in the calling service.

[0067] The method may additionally include creating a binding in the API registry between the API function and the corresponding endpoint of the service (705). Based on the discovery process described above and the registration process of steps 701, the API registry can now create a dynamic binding between an endpoint for a service in the container platform and the API function created by the API registry. In the data structure formed above when discovering available endpoints and services, the API registry can now store a corresponding function or set of functions for each endpoint. As described above, this binding can be constantly updated as

the discovery process determines when services are updated, moved, replaced, or added to the container platform. This allows the client libraries created in a calling service to first check with the API registry to verify or receive a current IP address and port number for the service.

[0068] **FIG. 7B** illustrates a hardware/software diagram of the steps for registering an API with the API registry 404, according to some embodiments. As described above, the API registry 404 can be instantiated and running in a container 620 in the container platform 210. Even though the container platform 210 represents a production/deployment environment, the API registry 404 can still be accessed by the IDE 102 used to develop the service. Thus, the IDE 102 can provide a mechanism for uploading the API definition files 702 to the API registry 404. Specifically, the user interface of the IDE 102 may include a window or interface that allows the developer to define and/or populate fields for the API definition files 702. This information described above may include function names, parameter lists, data types, field lengths, object class definitions, an IP address and port number, a service name or other unique identifier, and so forth. This information can be uploaded to the API registry 404 and linked in a dynamic binding to a particular IP address and port number for the endpoint 602. Finally, the API registry 404 can generate one or more API functions 704 that can be made available through the API registry 404.

[0069] After registering a service with the API registry 404 and generating one or more API functions, the API registry can then make those functions available for developers as they design services. **FIG. 8** illustrates examples of a graphical interface 802 and a command line interface 804 for browsing and selecting APIs that are registered with the API registry 804, according to some embodiments. When programming and developing a new service for the container platform, the developer can access the graphical interface 802 to browse and select API functions that can be used in their service. This graphical interface 802 is merely an example and not meant to be limiting of the types of graphical interfaces that can be used to browse and select API functions.

[0070] In this embodiment, the IDE 102 can summon the graphical interface 802 to provide a list of APIs that are registered with the API registry. In this embodiment, the APIs are categorized based on endpoint. For example, one endpoint corresponding to a service may offer a RESTful interface for storing user records (e.g., "UserStorage"). The graphical interface 802

can display all of the API functions (e.g., "CreateUser", "DeleteUser", "UpdateUser", etc.) that are available through the selected endpoint. Other embodiments may group functions based on the overall service in cases where the service offers multiple endpoints. The graphical interface 802 can receive a selection of one or more API functions to be used in a calling the service. The API registry can then provide documentation that illustrates how to use the API function, including required parameters and return values. One having ordinary skill in the art will understand that the command line interface 804 can provide similar information and can receive similar inputs as the graphical interface 802.

[0071] The interfaces 802, 804 illustrated in FIG. 8 provide a number of technical benefits. First, these interfaces 802, 804 provide an up-to-date listing of all APIs that are registered with the API registry. This corresponds to a list of all services currently available in the container platform. Instead of being required to look up documentation, contact a service developer, and/or perform other inefficient tasks for locating a list of available services, a service developer can retrieve and display this information in real-time. Additionally, as services are updated, the API definition files can be updated in a corresponding fashion. This then updates the display illustrated in FIG. 8 to provide up-to-date availability information for each API function.

[0072] **FIG. 9** illustrates a flowchart of a method for using a service and its corresponding function registered with the API registry, according to some embodiments. The method may include providing a listing of registered APIs (901). This step may be omitted in cases where the desired service is already known. However, generally the services can be displayed for browsing and navigation using the interfaces described above in FIG. 8. The method may also include receiving a selection of an API function (901). This selection may be received by the API registry from a developer of the service. For example, a developer may decide to update a database of user records using the CreateUser( ) function described above. **FIG. 10** illustrates how a selection 1002 may be received by the API registry through the graphical interface 802 for the CreateUser( ) function. Other embodiments may receive the selection through the command line interface or through other input methods provided by the IDE.

[0073] Referring back to FIG. 9, once the selection of an API function is received, the API registry can generate one or more client libraries for the calling service (905). Generating client libraries may provide the calling service with the service endpoint that is dynamically bound to

the API function. Specifically, the IDE can generate a set of class objects in the IDE that encapsulate the functionality required to interface directly with the service endpoint in the container platform. In some embodiments, client libraries may include object classes that can be instantiated or used to call member functions that embody the code required to communicate with the service. Examples of these client libraries will be described in greater detail below.

[0074] The method may additionally include providing test data (907). When a service is registered with the API registry, it need not be complete. Instead, the service can indicate to the API registry that it is not yet ready to provide functional responses to calling services. In some embodiments, the API definition file that is uploaded to the API registry can include a specification of the type of information that should be returned before the service is functional. When the calling service calls the API function, the client library generated by the API registry can route requests to the API registry instead of the service endpoint. The API registry can then provide a response using dummy, null, or default values. Alternatively, the code within the client libraries themselves can generate the default data to be returned to the calling service.

[0075] It should be appreciated that the specific steps illustrated in FIG. 9 provide particular methods of using an API registry according to various embodiments of the present invention. Other sequences of steps may also be performed according to alternative embodiments. For example, alternative embodiments of the present invention may perform the steps outlined above in a different order. Moreover, the individual steps illustrated in FIG. 9 may include multiple sub-steps that may be performed in various sequences as appropriate to the individual step. Furthermore, additional steps may be added or removed depending on the particular applications. One of ordinary skill in the art would recognize many variations, modifications, and alternatives.

[0076] **FIG. 11** illustrates an example of a client library generated for a service automatically by the API registry, according to some embodiments. This client library 1102 may correspond to a service that stores user records. This client library 1102 and the corresponding class and service are provided merely by way of example and not meant to be limiting. As described above, each API function and service can specify how client libraries should be generated by virtue of the API definition file uploaded to the API registry. Therefore, the principles described below in relation to the "User" service may be applied to other services.

[0077] To represent the User service, the API registry can generate a class for a User. When the calling service requests client libraries to be generated by the API registry, the calling service can specify a programming language being used by the calling service. For example, if the calling service is being written in Java in the IDE, then the API registry can generate class libraries in the Java programming language. Alternatively, if the calling service is being written in C#, then the API registry can generate class libraries in the C# programming language. The User class can be generated to have member functions that correspond to different operations that may be performed through the service endpoint. These member functions can be static such that they do not require an instantiated instance of the User class, or they may be used with instantiated User objects.

[0078] In this example, the User service may use a RESTful interface to edit individual user records that are stored by the service. For example, the API registry can generate the CreateUser( ) function to implement a POST call to the User service. One of the functions that can be performed by the class library is to parse, filter, and format data provided as parameters to the API function to be sent as a data packet directly to the service. In this example, the CreateUser( ) function can accept parameters that are formatted for the convenience of the calling service. For example, the calling service may separately store strings for the user first name and the user last. However, the POST command may require a concatenated string of the first name in the last name together. In order to accommodate a user-friendly set of parameters, the client library 1102 can perform a set operations that format the data received as parameters to the function into a format that is compatible with the service endpoint. This may include generating header information, altering the format of certain data fields, concatenating data fields, requesting addional data from other sources, performing calculations or data transforms, and so forth. This may also include packaging the reformatted parameters into a format, such as JSON, XML, etc.

[0079] Once the parameters are correctly formatted into a package for the service endpoint, the client library 1102 can also handle the POST call to the service. When the client library is generated, the IP address and port number for the service can be inserted into the CreateUser( ) function to be used in an HTTP request to the service. Note that the details of the HTTP request are encapsulated in the CreateUser( ) function. When a developer for a calling service wants to use the POST function made available by the service, instead of writing the code in the library

1102 themselves, they can instead select the User service from the API registry. The API registry will then automatically generate the client library 1102 that includes the User class. Then, to use the POST function, the service developer can simply use the User.CreateUser("John", "Smith", 2112) function to add the user John Smith to the service.

[0080] **FIG. 12** illustrates an embodiment of a client library 1202 that accommodates dynamic binding between service endpoints and API functions, according to some embodiments. In this example, when the API registry generates the client library 1202, the CreateUser( ) function can include code 1204 that dynamically retrieves the IP address and port number for the service. The calling service 114 can use the GetIPPort( ) function to send a request to the API registry 404 at run time when the calling service 114 is operating in the production/deployment environment 104, such as the container platform. The API registry 404 can access its internal table that is consistently updated to maintain up-to-date bindings between the API functions and the service endpoints. The API registry 404 can then return a current IP address and port number to the calling service 114. The client library 1202 can then insert the IP address and port number into the HTTP POST code that connects to the service. Because the API registry 404 can be accessed at run time by any calling service in the container platform, none of these services need to be updated or patched when the IP address for port number for the service being called changes. Instead, the API registry 404 can provide up-to-date information every time a service is called. In some embodiments, the GetIPPort( ) function may only need to call the API registry 404 once an hour, once a day, once a week, and so forth, to minimize the number of function calls made outside of the container for the service 114 under the assumption that the service endpoints do not change frequently in the production environment.

[0081] **FIG. 13** illustrates an embodiment of a client library 1302 that can marshal additional data to complete an input data set for a service call, according to some embodiments. To simplify using the client library 1302, the client library 1302 may minimize the number of parameters required from the service developer. Additional data that may be required to make the service call can be retrieved from other sources and thus may be omitted from the parameter list. These additional parameters can instead be retrieved directly by the client library 1302 from these other sources. For example, creating a new user may include specifying a user role for the user. Instead of requiring the service developer to provide a user role as one of the parameters, the client library 1302 can instead include code 1304 that automatically retrieves a role for the

user from some other source. In this example, the user role can be retrieved from a database, from another service in the container platform, or from another class storing user roles within the calling service. In any of these cases, the code 1304 can automatically retrieve the user role and package it as part of the input data for the HTTP POST command sent to the service.

[0082] In addition to marshaling and formatting data for inputs to the service, the client library 1302 can also parse and return data received from the service and handle error conditions. In this example, the POST command may return a data packet into the Result variable. Often times, a service may return a data packet that includes more information than the calling service needs. Therefore, the client library 1302 can parse the data fields in the Result variable and extract, format, and package data from the Result variable into a format that is more usable and expected by the User class. In this example, the code 1306 can extract fields from the Result variable and use them to create a new User object that is returned from the API function. In another example using a GET command, individual API functions can be generated in the User class that extract different fields from the Result variable from the GET command. For example, the User class could provide a GetFirstName(id) function, a GetLastName(id) function, a GetRole(id) function, and so forth. Each of these functions may include very similar code while returning different fields from the Result variable.

[0083] In addition to parsing results, the client library 1302 may also generate code 1308 that handles error conditions associated with using the service. In this example, the code 1308 can test a status field in the Result variable to determine whether the POST command was successful. If the command was successful, then the CreateUser( ) function can return a new User object. In cases where the Post command failed, the function can instead return a null object and/or retry the call to the service.

[0084] **FIG. 14** illustrates a client library 1402 that can handle retries when calling a service, according to some embodiments. Like the example of FIG. 13, the client library 1402 uses a status in a Result variable populated by the POST HTTP call to determine whether the call was successful or not. While the result is unsuccessful, the client library 1402 can continue to retry until the call is successful. Some embodiments may use a counter or other mechanism to limit the number of retries or add a wait time between retries.

[0085] As described above, some embodiments may also upload a set of API properties to the API registry along with the API definition. **FIG. 15A** illustrates a method of providing API properties to the API registry, according to some embodiments. The method may include receiving an upload of an API definition (1501). The method may also include receiving an upload of API properties (1503). The upload of properties may be part of the same transmission as the upload of the API definition. In some embodiments, the API properties may be part of the API definition. In some embodiments, the API properties may be one or more flags or predefined data fields that are checked to indicate that property should be set by the API registry. In some embodiments, the API properties need not conform to any pre-structured format, but can instead be represented by instruction code that causes the API registry to implement the features described below, such as authentication, encryption, and so forth. The API properties can be stored along with the API definition for each service.

[0086] The method may additionally include creating the API binding between the service and the API (1505). This operation may be performed as described in detail above. Additionally, the method may include using the API properties to perform one or more operations associated with the service (1507). The API properties may be used at different phases during the lifecycle of the service. Generally, this may be described as using the API properties to to implement a function associated with the property during the deployment of a service, when generating client libraries for service, and/or when calling service. Examples of each of these functions will be described below in greater detail.

[0087] It should be appreciated that the specific steps illustrated in FIG. 15A provide particular methods of providing API properties to an API registry according to various embodiments of the present invention. Other sequences of steps may also be performed according to alternative embodiments. For example, alternative embodiments of the present invention may perform the steps outlined above in a different order. Moreover, the individual steps illustrated in FIG. 15A may include multiple sub-steps that may be performed in various sequences as appropriate to the individual step. Furthermore, additional steps may be added or removed depending on the particular applications. One of ordinary skill in the art would recognize many variations, modifications, and alternatives.

[0088] **FIG. 15B** illustrates a hardware/software diagram of how a service can provide API properties to the API registry, according to some embodiments. While developing a service in the IDE 102, a service developer can provide the API definition file 1502 and one or more properties 1504 to the API registry 404. Because the API registry 404 is accessible in both the IDE 102 and the container platform at runtime, the API registry 404 can store the properties 1504 and use them to affect how a service is deployed, called, and/or used to generate client libraries during both development and runtime scenarios.

[0089] **FIG. 16** illustrates a hardware/software diagram where a property is used by the API registry to deploy a service with high availability, according to some embodiments. In addition to the API definition file 1505 for a particular service, the API registry 404 may receive a property 1602 indicating that the service should be deployed to be very resilient, or have high availability. This property 1602 may be received as a set of instructions that are executed by the API registry 404 to deploy the service to have high availability. This option allows the developer to define what it means to be "high-availability" for this service. For example, the property 1602 may include instructions that cause the API registry 404 to deploy multiple instances 602, 604 of the service to the container platform 210. By executing these instructions, the API registry 404 does not need to make any decisions or determinations on its own, but can instead simply execute the deployment code provided as part of the property 1602.

[0090] The property 1602 may also be received as a flag or setting that indicates to the API registry 404 an option to execute existing instructions at the API registry 404 for deploying the service with high availability. With this option, the API registry 404 need not receive any code to be executed as the property 1602. Rather, the API registry 404 can recognize the high-availability property 1602 and execute code that is maintained in the API registry 404 to deploy multiple instances 602, 604 of the service. This allows the API registry 404 to define what it means to be "high-availability" for the deployment of any service that is registered with the API registry 404.

[0091] Because the API registry 404 is connected to the runtime environment of the container platform 210, the API registry 404 can interact with the container platform 210 to deploy the instances 602, 604 that determine the runtime availability of the service. Note that the two instances 602, 604 of the service illustrated in FIG. 16 are provided merely as an example and

not meant to be limiting. A high-availability service may include more than two redundant instances of a service being deployed to the container platform.

[0092] Some embodiments may include code in the API registry 404 that can be executed as a default. If the property 602 includes only a simple indication that high availability is desired, the API registry 404 can execute its own code. If the property 602 includes deployment code for deploying the service, the API registry 404 can instead execute the code of the property 1602. In some cases, the property 1602 may include only a portion of the code needed to deploy the service with high-availability. The API registry 404 can execute the portions of the code that are provided by the property 1602, then execute any code not provided by the property 1602 using the code at the API registry 404. This allows developers to overwrite an existing definition of how to execute a property, such as high-availability, at the API registry 404, while still allowing the API registry 404 to provide a uniform definition for executing properties that can be used by registered services.

[0093] In some embodiments, a high-availability property may also cause the container platform 210 to deploy a load balancing service 606 that distributes requests to the multiple instances 602, 604 of the service. The endpoint of the load balancing service 606 can be registered with the API registry 404 and made available to other services. Alternatively or additionally, each of the multiple instances of the service 602, 604 may be registered with the API registry 404 as service endpoints.

[0094] In each of the examples described below, the same principles discussed in relation to FIG. 16 may apply. For example, any property described below may be accompanied with code that may be received by the API registry 404 and used to overrule code that would otherwise be executed by the API registry 404. Prior to this disclosure, no method existed for creating a uniform default for executing properties while simultaneously allowing service developers to overrule those properties if needed. Therefore, the API registry 404 solves a technical problem by allowing code to be executed at the API registry 404 as a default while still allowing that code to be overruled by a property 1602 received from a developer.

[0095] **FIG. 17** illustrates a hardware/software diagram of a property that enforces end-to-end encryption through the API registry, according to some embodiments. Along with the API definition file 1505, the API registry 404 may receive a property 1704 that indicates, or includes

code that generates, end-to-end encryption for calling the service 1708. During development, the service 1708 can include its own decryption/encryption code 1710 that causes packets received by the service 1708 to be decrypted and packets returned by the service 1708 to be encrypted. Prior to this disclosure, the developer would need to provide a specification that indicated users of the service 1708 needed to provide encryption to be compatible with the service 1708. This embodiment solves a technical problem by allowing the service 1708 to dictate how the client libraries are generated in a calling service 1706, which ensures compatibility with the encryption of the service 1708.

[0096] In some embodiments, the developer of the service 1708 need not include the encryption/decryption code 1710 in the service 1708. Instead, the property 1704 can simply instruct the API registry 404 to enforce end-to-end encryption for the service 1708. When the service 1708 is deployed to the container platform 210, the API registry 404 can cause the encryption/decryption code 1710 to be inserted into the service 1708 when it is deployed. This allows the developer to select between different encryption regimes based on the property 1704 and/or to allow the API registry 404 to select a preferred encryption regime as a default.

[0097] End-to-end encryption requires not only the encryption/decryption code 1710 to be inserted into the service 1708 when it is deployed or during development, but it also requires that a calling service 1706 also includes compatible encryption/decryption code. As described above, when the calling service 1706 needs to use the service 1708, the API registry 404 can generate one or more client libraries 1702 that completely implement the code needed to interact with the service 1708 in a simple and efficient manner. When this client library 1702 is generated, the API registry 404 can analyze the property 1704 to determine an encryption regime used by the service 1708. Then, based on that property 1704, the API registry 404 can cause a compatible encryption/decryption code to be added to the client library 1702 for the calling service 1706. Thus, when the calling service 1706 sends a request to the service 1708, the information may be encrypted at the calling service 1706 and decrypted once received by the service 1708. Similarly, the service 1708 can encrypt a response before it is sent to the calling service 1706, which can then decrypt the response before passing the response outside of the client library 1702. This causes the entire encryption process to be entirely transparent to a developer of the calling service 1706. Instead of being required to implement a compatible encryption/decryption regime when calling the service 1706, the property 1704 may ensure that the API registry 404

has already generated the encryption/decryption code in the client library 1702 to be compatible and implement the end-to-end encryption property.

[0098] **FIG. 18** illustrates a property 1804 for an API registry to implement usage logging for a service 1808, according to some embodiments. Prior to this disclosure, to monitor and log the frequency, source, success rate, etc., of requests to a service, the service itself had to log this information. Alternatively, the container environment had to monitor the service and log its usage information. Logging information at the service 1808 itself is terribly inefficient, and slows down the throughput for every request handled by the service. Similarly, the overhead of requiring the container platform to monitor and log all the calls made to particular services also represents a tremendous overhead to the scheduling and orchestration of container services. This embodiment solves this technical problem by inserting code directly into client libraries for services that call the service 1808. This allows the usage of the service 1808 to be logged and monitored without affecting the performance of the service 1808 at all in terms of memory usage or CPU usage.

[0099] In addition to the API definition file 1505, the API registry 404 can receive a property 1804 that indicates, or includes code that implements, usage logging 1804. When a developer of a calling service 1806 desires to submit requests to the service 1808, the API registry 404 can automatically generate a client library 1802 that includes code for logging activity related to the service 1808. As described above, this code can be generated based on default code maintained and executed by the API registry 404, or can be generated by code received with the property 1804 and executed by the API registry 404.

[0100] The code for logging activity in the client library 1802 may include counters that are incremented every time the service 1808 is called, functions that cause activity to be logged to a log file when the service 1808 is called, and other functions that monitor and record characteristics of the requests sent to the service 1808 and responses received from the service 1808. Depending on the particular embodiment, this code may monitor many different types of characteristics associated with requests made of the service 1808. For example, some embodiments may log the total number of calls made to the service 1808. Some embodiments may log a success rate for responses received from the service 1808. Some embodiments may log types of data that are sent in requests to the service 1808. Some embodiments may log times

of day or other external information for when the service 1808 is called. Some embodiments may log input and output packets to/from the service 1808 that can be used for debugging the service 1808. Some embodiments may log any or all of these characteristics in any combination and without limitation.

[0101] **FIG. 19** illustrates a hardware/software diagram of a property 1904 that can enforce an authentication protocol for a service 1908, according to some embodiments. Some services may require that a user identity be authenticated and that the user be authorized to use the service before responding to a request. Prior to this disclosure, a technical problem existed where the authentication and authorization procedures took place at the service 1908 itself. This added overhead in terms of memory usage and CPU usage for every call received by the service 1908, and increased the latency of the service in response. This in turn decreased throughput, and limited the number of requests that could be processed by the service 1908 during any given time interval. These embodiments solve this technical problem by moving authentication/authorization code to the client library 1902 that is automatically generated by the API registry 1404.

[0102] When a calling service 1906 wants to use the service 1908, the API registry 404 can generate the client library 1902 that includes code for performing the authorization and/authentication. In some embodiments, this may include contacting external authentication/authorization services 1920 that specifically verify user identities and/or determine whether a user is authorized to use the service 1908. The external authentication/authorization services 1920 may include an access manager, a Lightweight Directory Access Protocol (LDAP) manager, an Access Control List (ACL), a network authentication protocol manager, and so forth. The code in the client library 1902 can then send the call to the service 1908 when the authentication/authorization procedure is successful.

[0103] By offloading the authentication/authorization enforcement to the API registry 404 and the client library 1902, this code can be completely eliminated from the service 1908. Because significant delays may often accompany interacting with the external authentication/authorization services 1920, this delay can be removed from the service 1908 to increase throughput. Additionally, rather than hard coding the authentication/authorization enforcement into the service 1908, the developer of the service 1908 can instead simply select a

predefined authentication/authorization regime using the property 1904 that is sent to the API registry 404. The API registry 404 can maintain a predefined list of authorization/authentication with the accompanying implementation code for the client library 1902. This also prevents the calling service 1906 from sending requests to the service 1908 that cannot be authorized and/or authenticated. Instead, if the authentication and/or authorization routine is unsuccessful, the call can be aborted at the client library 1902. This ensures that the service 1908 only receives requests that are authenticated and/or authorized.

[0104] Another technical improvement provided by the API registry 404 is the ability to upgrade any of the functionality provided by the properties 1904 without being required to change any of the code of any registered services. For example, because the authentication/authorization code has been offloaded to the client library 1902 generated by the API registry 1404, the client library 1902 can be updated to change the authentication/authorization regime. None of the code in the calling service 1906 or the service 1908 needs to be modified. Because code is only changed in a single place, this greatly reduces the probability of code integration errors that would otherwise accompany distributed patches sent out to every individual service.

[0105] **FIG. 20** illustrates a hardware/software diagram for a property 2004 that enables runtime instantiation of a service, according to some embodiments. Some services may be rarely used or only used during predefined time intervals. Therefore, deploying a service to the container platform need not always result in actually instantiating an instance of the service in a container that is immediately available. In contrast to virtual machines, containers can be instantiated and activated very quickly. Therefore, a service developer may desire to only have the service instantiated when it is called. A service developer may also desire to only have the service instantiated within a predefined time interval. Similarly, the service developer may specify that the service instance should be deleted after a predefined time interval of inactivity.

[0106] In addition to receiving the API definition file 1505, the API registry 404 can receive a property 2004 that specifies run-time instantiation or other instantiation parameters. For example, the property may include a specification of one or more time intervals during which the service 2008 should be instantiated after deployment. In another example, the property may include an indication that the service 2008 should only be instantiated on demand. In another

example, the property may specify a timeout interval after which the instantiated service 2008 should be deleted from the container platform.

[0107] When a calling service 2006 wants to use the service 2008, the API registry 404 can generate code in the client library 2002 that handles the run-time instantiation of the service 2008. For example, the CreateInstance( ) function call in the client library 2002 can create a call to the API registry 404. The API registry can then interact with the container platform 210 to determine whether an operating instance of the service 2008 is available. If not, the API registry 404 can instruct the container platform 210 to instantiate an instance of the service 2008 in a container in the container platform 2010. The container platform 210 can then return the endpoint (e.g., IP address and port number) to the API registry 404. The API registry 404 can then create a binding between that endpoint and the API function call created in the client library 2002. API registry 404 can then return the endpoint to the client library 2002 which can be used to create the direct connection between the calling service 2006 and the newly instantiated service 2008.

[0108] For services that should only be instantiated during predefined time intervals, the API registry 404 may establish a table of instantiation and deletion times for certain services. Based on these stored instantiation/deletion times, the API registry 404 can instruct the container platform 210 to instantiate or delete instances of the service 2008. The API registry 404 can also specify a number of instances that should be instantiated during these predefined intervals. For example, from 5:00 PM to 10:00 PM the property 2004 may specify that at least 10 instances of the service 2008 are active on the container platform 210. When this time interval occurs, the API registry 404 can instruct the container platform 210 to create the additional instances.

[0109] **FIG. 21** illustrates a hardware/software diagram of a property 2104 that implements a rate limiting function for a service 2108, according to some embodiments. Some services may need to limit the rate at which requests are received. Other services may need to limit requests from certain senders or types of services. Prior to this disclosure, this function had to be performed by the service itself by determining a source for each request, comparing the source to a whitelist/blacklist, and throttling the rate at which it serviced these requests. As with most of the examples described above, placing this overhead in the service itself increase the amount of memory and CPU power used by the service and limited the throughput of the service. These

embodiments solve this technical problem by automatically generating the rate limiting code in the client library generated by the API registry. This allows the service to specify rate limiting by virtue of the property 2104 without requiring the service 2108 to implement that functionality with all of its associated overhead.

[0110] When a calling service 2106 wants to send requests to the service 2108, the API registry 404 can automatically generate the client library 2102 that includes rate limiting code. When the client library 2102 is generated, the API registry 404 can determine whether the particular service 2106 should be rate limited. If not, the client library 2102 can be generated as usual. If the API registry 404 determines that the calling service 2106 should be rate limited (e.g., by comparison to a whitelist/blacklist), then the API registry 404 can insert code in the client library 2102 that adds delays, adds counters, and/or otherwise implements the rate limiting function to ensure that a predefined maximum number of requests are made by the calling service 2106 in any given time interval according to a predefined rate. This code may also implement time windows during which the rate limiting function will be active. This allows the service 2108 to enforce rate limiting during high-traffic intervals automatically.

[0111] As described above, the system may include an IDE 102 and a production/deployment environment 104. The production/deployment environment 104 may include a cloud computing platform 202. Applications and services can be developed in the IDE 102 and deployed to the cloud computing platform 202. Typically, the IDE 102 will include a debugger 110. A debugger 110 is a software program that is specifically designed to test for errors and locate bugs in services or applications as they are being developed. Debuggers may use instruction-set simulators or may run a program directly on the underlying hardware/processor to achieve a high level of control over the execution of the program instructions. Most importantly, the debugger allows the developer to stop or halt program execution according to specific conditions represented by breakpoints. While the program execution is stopped, the developer can examine variable values, program execution flows, and/or other characteristics of the state of the application or service.

[0112] While a debugger 110 is particularly useful during the development process in the IDE 102, it is typically not deployed with the service into the cloud computing platform 202. Deploying a debug version of the service would introduce technical problems that would prevent

the efficient use of the cloud computing platform 202. For example, a version of the service that is compatible with the debugger, referred to herein as a "debug build" typically includes unnecessary overhead and additional instructions that are required for compatibility with the debugger 110, but which are not required for regular execution. These additional instructions/overhead cause the debug build of the service to take longer to run, which increases latency, reduces throughput, and uses more processing resources of the cloud computing platform 202. Additionally, the debug build would need to be deployed with the debugger 110 software. Deploying a version of the debugger 110 with every instance of the service would add a tremendous amount of additional memory usage for each instance of the service, which can in turn reduce the available memory resources of the cloud computing platform 202.

[0113] Because the debugger 110 is not deployed into the production/deployment environment 104, troubleshooting errors that are uncovered after deployment is particularly difficult. Prior to this disclosure, a debug build of the service with the debugger 110 would be run in the IDE using dummy data or preconfigured inputs to try and simulate the real-world flow of request traffic that the service receives in the production/deployment environment 104. However, this prevents the service from using real-time, live data that may be optimal for re-creating errors and determining whether fixes/patches for these errors work properly without interfering with the expected operation of the service.

[0114] The embodiments described herein make changes to the cloud computing platform 202 such that a debug build of the service can be deployed with a debugger in the production/deployment environment 104 to receive real-time, live data from actual request traffic. In some embodiments, a debug client can send specific debug requests that are routed through a load balancer and directed by a service mesh to a debug instance of the service. This allows requests from other clients to pass through the same load balancer and service mesh while continuing to be routed directly to normal instances of the service. In some embodiments, the service mesh can receive requests from any client and clone the requests to be sent to the debug instance of the service. These embodiments allow a debug instance of the service to operate in the cloud computing platform 202 of the production/deployment environment 104 alongside the normal instances of the service. These embodiments also solve the technical problems described above by routing real-time, live requests to the debug instance of the service without interrupting the normal operations of other service instances.

[0115] **FIG. 22** illustrates a block diagram of a portion of the cloud computing platform 202 for receiving service requests, according to some embodiments. As described above, some embodiments may include an API registry 404 that can govern how services are deployed and operated within the cloud computing platform 202. The API registry 404 can be communicatively coupled to a service mesh 2210. A service mesh 2210 is a configurable infrastructure layer for a microservice or service-oriented environment that allows communication between service instances to be flexible, reliable, and fast. In some embodiments, the service mesh 2210 may provide service discovery, load-balancing, encryption, authentication/authorization, and other capabilities. In the cloud computing platform 202, as more containers are added to the infrastructure, the service mesh 2210 routes requests from clients to specific instances of each service. Note that the API registry 404 is not required in all embodiments. In some embodiments, the functions described below may also be performed by a scheduler or other component of a container orchestration layer.

[0116] In some embodiments, the service mesh 2210 may provide its own load-balancing operations. For clarity, FIG. 22 exclusively illustrates a load balancer 2208 which may be part of the service mesh 2210 or may be a separate component. The load balancer 2208 can be used to route requests from a plurality of clients 2202, 2204, 2206 to different service instances. The different service instances may reside in different operating environments, different container nodes, and/or on different hardware devices. Additionally, a load balancer 2212 may be associated with a specific service. A single service may have a plurality of service instances 2214, 2216, 2218 that are available for servicing requests. The load balancer 2212 can receive requests from the service mesh and route request traffic to specific service instances 2214, 2016, 2218 such that the service requests are evenly distributed amongst the service instances 2214, 2016, 2218.

[0117] In the example of FIG. 22, each of the service instances 2214, 2016, 2218 has been deployed to the cloud computing platform 202 as a production version of the service. These may be referred to herein as "production services" or "normal services," and they may be distinguished from debug instances of the service in that the containers/pods for the service instances 2214, 2016, 2218 do not include debug builds of the services or debuggers. Instead, the these "normal" service instances 2214, 2016, 2218 have been compiled and deployed in a

streamlined format that allows them to quickly and efficiently service requests without the overhead associated with debugging operations.

[0118] **FIG. 23** illustrates a debug build of a service encapsulated in a pod 2300, according to some embodiments. The pod 2300 is similar to the pod illustrated in FIG. 3. For example, the overall service is available through an endpoint 318, and the service is constructed using a plurality of microservices or services 320, 322, 324, 326. These services may be packaged in a plurality of containers 310, 312 and may make use of a common resource 308, such as a storage volume, a lookup table, a database, or other resource.

[0119] However, in contrast to the pod 304 in FIG. 3, this debug build of the service encapsulated in pod 2300 includes additional debugging utilities. For example, the pod 2300 includes an additional container 2302 that includes a debug daemon 2304. In general, a daemon is a computer program that runs as a background process rather than being under the direct control of an interactive user. In this specific container environment, the debug daemon 2304 comprises a background process that runs outside of the control of the service itself and is configured to interface with the service instance. In addition to the debug daemon, the container 2302 may include a debug log in service 2308 that allows an administrator to login to the pod 2300 and perform debugging operations while it is operating. The container 2302 may also include additional debug utilities 2006, such as utilities that track memory usage, processor usage, bandwidth usage, and other computing characteristics while the service receives real-time, live data requests.

[0120] In addition to the container 2302, the pod 2300 may include a specific debug build of the service itself. For example, one or more of the services 320, 322, 324, 326 may be built or compiled using a debug configuration such that the source code of the services 320, 322, 324, 326 includes interfaces for a debugger process 2310 to operate with the debug build of the service. One or more of the services 320, 322, 324, 326 would thus be able to provide variable values and pause execution based on interactions with the debugger 2310. The operation of the service itself may be considered a clone of the other normal instances of the service 2214, 2216, 2218 from FIG. 22. Thus, the debug build of the service illustrated in the pod 2300 of FIG. 23 can be used to reliably debug the normal instances of the service 2214, 2216, 2218 from FIG. 22.

[0121] **FIG. 24** illustrates an alternative pod 2400 for a debug build of a service, according to some embodiments. Pod 2400 is similar to pod 306 in FIG. 3. Specifically, pod 2400 includes a single container 314 with a service 328 provided through an endpoint 316. However, the container 314 may also be loaded with the debugging utilities needed to provide for a debug build of the overall service. Specifically, the service 328 can be compiled using a debug configuration and loaded into the container 314. Additionally, the debug daemon 2304, debug login 2308, debugger 2310, and/or additional debug utilities 2306 may also be included in the container 314 in any combination and without limitation. It will be understood that none of these debug utilities is required for a specific embodiment, and therefore some embodiments may omit any of the debug utilities described above.

[0122] In the description below, the debug instances of the service described above in FIG. 23 in FIG. 24 can be instantiated in the production/deployment environment 104 during operation. For example, after the normal service instances 2214, 2016, 2218 have been instantiated and operated in the cloud computing platform 202, a debug instance of the service may also be instantiated and loaded into the cloud computing platform 202. The debug instance of the service may receive regular and/or specialized request traffic in a manner similar to that of the normal service instances 2214, 2216, 2218. This allows real transactions to be debugged in a production environment in real-time without interrupting the regular operation of the normal service instances 2214, 2216, 2218.

[0123] **FIG. 25** illustrates a block diagram of a system for instantiating a debug instance of a service 2502, according to some embodiments. The cloud computing platform 202 can receive normal requests from regular clients 2204, 2206. These normal requests can be routed through the load balancer 2208 and the service mesh 2210 to a particular service. That service may have multiple instances 2214, 2216, 2218, and a load balancer 2212 for that service can route requests such that the load at each service instance is balanced.

[0124] In addition to receiving normal requests from the regular clients 2204, 2206, some embodiments may also receive debug request 2506 from a debug client 2504. The debug client 2504 may be identical to the regular clients 2204, 2206, and thus may comprise a desktop computer, laptop computer, workstation, server, mobile computing device, smart phone, PDA, tablet computer, laptop computer, smart watch, and so forth. However, the debug client 2504

may be specifically identified as a client device that provides debug messages to the cloud computing platform 202. A debug request 2506 may be identical to normal requests sent from the regular clients 2204, 2206. However the debug request 2506 may be specifically provided by the debug client 2504 to go to a debugging instance of the service rather than the normal instances of the service 2214, 2216, 2218.

[0125] When the service mesh 2210 receives the debug request 2506, it can determine whether or not a debugging instance of the service is currently operating in the container platform 210. If there is no debug instance of the service currently operating on the container platform 210, the service mesh 2210 can send a message to the API registry 404 to generate alternatively, the service mesh 2210 can initiate a message to a scheduler for the cloud computing platform 202 to generate a debugging instance of the service in embodiments that do not use the API registry 404.

[0126] As described above, registering the service with the API registry 404 can be accomplished by providing an API definition file 1505 for the service. The API definition file 1505 can also be accompanied by one or more properties that characterize how a service should be deployed and/or how client libraries should be generated for calling services. In this case, the one or more properties may include a dynamic debugging property 2510 that specifies to the API registry 404 that a debug instance of the service should be generated on demand when a debug request 2506 is recognized or identified by the service mesh 2210. Normal instances of the service 2214, 2216, 2218 can be deployed without being affected by the dynamic debugging property 2510. Prior to receiving the debug request 2506, no debug instances of the service need to be instantiated or operated on the container platform 210.

[0127] After deployment of the normal instances of the service 2214, 2216, 2218, and in response to receiving the debug request 2506, the API registry 404 can cause a debugging instance of the service 2502 to be deployed in the cloud computing platform 202. The debug instance 2502 can be a single container instance as illustrated in FIG. 24 or may be a multi-container instance as illustrated in FIG. 23. In either embodiment, the debug instance may include a debug build of the service, a debugger, a debug login, a debug daemon, and/or any other debugging utilities in any combination and without limitation.

[0128] As with any service that is deployed to the cloud computing platform 202, the debug instance of the service 2502 can be registered with the API registry 404. This may include providing an endpoint (e.g., an IP address and port number) to the API registry 404. The API registry 404 can then provide a set of API functions that can be used to call the debug instance of the service 2502 as described above. Alternatively or additionally, the debug instance of the service 2502 need not provide its own API definition file or properties to the API registry 404 since the API registry 404 has deployed the debug instance of the service 2502. Instead, the API registry 404 may provide the endpoint to the service mesh 2210. The service mesh 2210 can then use the endpoint to route the debug message 2506 to the debug instance of the service 2502. This endpoint does not need to be published because only the service mesh 2210 needs to send any messages to the debug instance. However, some embodiments may make this endpoint available to other services that access the debug information during runtime.

[0129] **FIG. 26** illustrates a block diagram of the container platform 210 routing debug requests to the debug instance of the service 2502, according to some embodiments. When the debug request 2506 is received by the service mesh 2210, the service mesh 2210 can identify the debug request 2506. The debug request 2506 may be distinguished from normal requests from the regular clients 2204, 2206 in a number of different ways. For example, the service mesh 2210 can identify the debug request 2506 by virtue of a sender, namely the debug client 2504. When the service mesh 2210 recognizes an address of the debug client 2504, the service mesh 2210 can designate the request as a debug request 2506. This allows the service mesh 2210 to identify debug requests by virtue of their source.

[0130] In some embodiments, the service mesh 2210 can identify the debug request 2506 using any information embedded within the debug request 2506. For example, the debug request 2506 may have unique header information or one or more flag settings within the debug request 2506 that identify it as a debug request. In another example, the debug request 2506 may include a payload within the request that includes a predetermined value that is only used by debug requests.

[0131] When the service mesh identifies the debug request 2506, it can change the normal routing of the debug request. Whereas normal requests can be sent to the load balancer 2212 and/or the normal instances of the service 2214, 2216, 2218, the debug request 2506 can be

specially routed to the debug instance of the service 2502. For example, the debug request 2506 from FIG. 25 can be used first to trigger the instantiation of the debug instance of the service 2502. When the service mesh 2210 receives a notification from the API registry 404 that the debug instance of the service 2502 is operational, the service mesh 2210 can then route the debug request 2506 to the debug instance of the service 2502. When future debug requests are received from the debug client 2504 or other clients designated as debug clients, the service mesh 2210 can also route these requests to the debug instance of the service 2502. Requests may thus be routed without additional participation from the API registry 404 because the debug instance of the service 2502 is already operational.

[0132] In some embodiments, the debug instance of the service 2502 can remain instantiated for a predetermined time limit. For example, the debug instance of the service 2502 can include a default time limit (e.g., 24 hours), after which the API registry 404 and/or the scheduler can delete the debug instance of the service 2502. In some embodiments, the dynamic debugging property 2510 can also specify a time interval that is tailored specifically for that service. For example, a service developer may determine that one week is a more appropriate debugging interval for a particular service, and a seven-day interval can then be provided as part of the dynamic debugging property 2510 to override any default time interval that would otherwise be applied by the API registry 404. In some embodiments, the time limit may be absolute such that it begins with instantiation. In some embodiments the time limit may be relative, such that it is reset each time a new debug request 2506 is received by the debug instance of the service 2502.

[0133] **FIG. 27** illustrates a block diagram of a cloud computing platform 202 that clones requests, according to some embodiments. In contrast to the embodiments described above, these embodiments do not require a special request source to provide debug requests to a debug instance of the service. Instead, the service mesh 210 can receive normal requests from regular clients 2202, 2204, 2206. Instead of routing special debug messages to the debug instance of the service in real time alongside the live data, these embodiments can use the live data itself and route cloned copies of the requests to the debug service without disrupting the normal operation of the normal instances of the service 2214, 2216, 2218.

[0134] As described above, the service itself can register with the API registry 404 by uploading an API definition file 1505. Along with the API definition file 1505, the API registry

404 can receive an upload of a clone debugging property 2710. The clone debugging property 2710 can indicate to the API registry 404 that a debug instance of the service 2502 should be instantiated in the cloud computing platform 202. The API registry 404 and/or the scheduler can deploy the debug instance of the service 2502 when the normal instances of the service 2214, 2216, 2218 are deployed. Alternatively or additionally, the debug instance of the service 2502 can be deployed in response to receiving a request from a client that is sent to the service. For embodiments that use the API registry 404, the API registry can provide the endpoint for the debug instance of the service 2502 to the service mesh 2210.

[0135] The API registry 404 and/or the scheduler can also provide an indication 2702 to the service mesh 2210 indicating that requests sent to the service should be cloned and also sent to the debug instance of the service 2502. The indication 2702 may indicate that all requests should be cloned and sent to the debug instance of the service 2502. In some embodiments, the indication 2702 may indicate that a certain percentage of requests should be cloned for the debug instance of the service 2502 (e.g., every other request, every third request, and so forth).

[0136] **FIG. 28** illustrates a block diagram of cloned requests being forwarded to a debug instance of the service 2502, according to some embodiments. When a normal request 2806 is sent from a regular client 2202, the normal request 2806 can go through the load balancer 2208 and be identified by the service mesh 2210. The service mesh 2210 can identify the normal request 2806 as being addressed to the service. Based on the indication 2702 and/or this identification, the service mesh 2010 can then generate a cloned request 2808. In some embodiments, the cloned request 2808 may be a complete copy of the normal request 2006. In some embodiments, the cloned request 2808 may include additional information that is inserted by the service mesh 2210 that may be useful in the debugging process (e.g., timestamps, or other diagnostic information that may be useful in a debugging scenario).

[0137] The cloned request 2808 can then be forwarded to the debug instance of the service 2502. This can be done in parallel with the normal request 2806 being forwarded to a normal instance of the service 2214, 2216, 2218. Thus, this cloning process can operate without affecting the throughput of the normal instances of the service 2214, 2016, 2218. This also allows the debug instance of the service 2502 to use real-time, live data for the debugging process. Thus, any error that occurs in the normal instances of the service 2214, 2216, 2218 will

also be captured by the output of the debug instance of the service 2502. This provides the unique technical advantage that allows problems to be isolated in real time as they occur using live debug data without the overhead that would normally accompany processing the normal request 2806 by a debug instance of the service exclusively.

[0138] **FIG. 29** illustrates a flowchart of a method for providing runtime debugging for containerized services in container environments. The method may include receiving a request for service at a container environment (2902). The container environment may include a service mesh and a plurality of services encapsulated in a plurality of containers. The service may also be encapsulated in a first one or more containers in the container environment. The container environment may include the container platform 210 described above. The request may be a debug request or a normal request from a regular client device. The first one or more containers may be organized into a container pod, which may include one or more microservices that form the service. The container environment may include an orchestrated container platform with a container scheduler. The container environment may also include an API registry.

[0139] The method may also include determining that a request should be routed to a debugging instance of the service (2904). This determination may be made based on a number of different factors, including a source of the request, a header in the request, values in the payload of the request, a flag in the request, timing of the receipt of the request, and/or any other characteristic of the request. If the request is not a debug request, then the method may include routing the request to a normal instance of the service (2908).

[0140] If the request is identified as a debug request, then the method may further include determining whether a debug instance of the service is available (2906). If the debug instance is available, then the method may include routing the request to the debug instance of the service (2912). If the debug instance is not available, then the method may also include instantiating the debug instance of the service (2910), and routing the request to the newly instantiated debug instance of the service (2912).

[0141] It should be appreciated that the specific steps illustrated in FIG. 29 provide particular methods of enabling live debugging in a container environment according to various embodiments of the present invention. Other sequences of steps may also be performed according to alternative embodiments. For example, alternative embodiments of the present

invention may perform the steps outlined above in a different order. Moreover, the individual steps illustrated in FIG. 29 may include multiple sub-steps that may be performed in various sequences as appropriate to the individual step. Furthermore, additional steps may be added or removed depending on the particular applications. One of ordinary skill in the art would recognize many variations, modifications, and alternatives.

[0142] Each of the methods described herein may be implemented by a specialized computer system. Each step of these methods may be executed automatically by the computer system, and/or may be provided with inputs/outputs involving a user. For example, a user may provide inputs for each step in a method, and each of these inputs may be in response to a specific output requesting such an input, wherein the output is generated by the computer system. Each input may be received in response to a corresponding requesting output. Furthermore, inputs may be received from a user, from another computer system as a data stream, retrieved from a memory location, retrieved over a network, requested from a web service, and/or the like. Likewise, outputs may be provided to a user, to another computer system as a data stream, saved in a memory location, sent over a network, provided to a web service, and/or the like. In short, each step of the methods described herein may be performed by a computer system, and may involve any number of inputs, outputs, and/or requests to and from the computer system which may or may not involve a user. Those steps not involving a user may be said to be performed automatically by the computer system without human intervention. Therefore, it will be understood in light of this disclosure, that each step of each method described herein may be altered to include an input and output to and from a user, or may be done automatically by a computer system without human intervention where any determinations are made by a processor. Furthermore, some embodiments of each of the methods described herein may be implemented as a set of instructions stored on a tangible, non-transitory storage medium to form a tangible software product.

[0143] **FIG. 30** depicts a simplified diagram of a distributed system 3000 that may interact with any of the embodiments described above. In the illustrated embodiment, distributed system 3000 includes one or more client computing devices 3002, 3004, 3006, and 3008, which are configured to execute and operate a client application such as a web browser, proprietary client (e.g., Oracle Forms), or the like over one or more network(s) 3010. Server 3012 may be

communicatively coupled with remote client computing devices 3002, 3004, 3006, and 3008 via network 3010.

[0144] In various embodiments, server 3012 may be adapted to run one or more services or software applications provided by one or more of the components of the system. In some embodiments, these services may be offered as web-based or cloud services or under a Software as a Service (SaaS) model to the users of client computing devices 3002, 3004, 3006, and/or 3008. Users operating client computing devices 3002, 3004, 3006, and/or 3008 may in turn utilize one or more client applications to interact with server 3012 to utilize the services provided by these components.

[0145] In the configuration depicted in the figure, the software components 3018, 3020 and 3022 of system 3000 are shown as being implemented on server 3012. In other embodiments, one or more of the components of system 3000 and/or the services provided by these components may also be implemented by one or more of the client computing devices 3002, 3004, 3006, and/or 3008. Users operating the client computing devices may then utilize one or more client applications to use the services provided by these components. These components may be implemented in hardware, firmware, software, or combinations thereof. It should be appreciated that various different system configurations are possible, which may be different from distributed system 3000. The embodiment shown in the figure is thus one example of a distributed system for implementing an embodiment system and is not intended to be limiting.

[0146] Client computing devices 3002, 3004, 3006, and/or 3008 may be portable handheld devices (e.g., an iPhone®, cellular telephone, an iPad®, computing tablet, a personal digital assistant (PDA)) or wearable devices (e.g., a Google Glass® head mounted display), running software such as Microsoft Windows Mobile®, and/or a variety of mobile operating systems such as iOS, Windows Phone, Android, BlackBerry 10, Palm OS, and the like, and being Internet, e-mail, short message service (SMS), Blackberry®, or other communication protocol enabled. The client computing devices can be general purpose personal computers including, by way of example, personal computers and/or laptop computers running various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems. The client computing devices can be workstation computers running any of a variety of commercially-available UNIX® or UNIX-like operating systems, including without limitation the variety of

GNU/Linux operating systems, such as for example, Google Chrome OS. Alternatively, or in addition, client computing devices 3002, 3004, 3006, and 3008 may be any other electronic device, such as a thin-client computer, an Internet-enabled gaming system (e.g., a Microsoft Xbox gaming console with or without a Kinect® gesture input device), and/or a personal messaging device, capable of communicating over network(s) 3010.

[0147] Although exemplary distributed system 3000 is shown with four client computing devices, any number of client computing devices may be supported. Other devices, such as devices with sensors, etc., may interact with server 3012.

[0148] Network(s) 3010 in distributed system 3000 may be any type of network familiar to those skilled in the art that can support data communications using any of a variety of commercially-available protocols, including without limitation TCP/IP (transmission control protocol/Internet protocol), SNA (systems network architecture), IPX (Internet packet exchange), AppleTalk, and the like. Merely by way of example, network(s) 3010 can be a local area network (LAN), such as one based on Ethernet, Token-Ring and/or the like. Network(s) 3010 can be a wide-area network and the Internet. It can include a virtual network, including without limitation a virtual private network (VPN), an intranet, an extranet, a public switched telephone network (PSTN), an infra-red network, a wireless network (e.g., a network operating under any of the Institute of Electrical and Electronics (IEEE) 802.11 suite of protocols, Bluetooth®, and/or any other wireless protocol); and/or any combination of these and/or other networks.

[0149] Server 3012 may be composed of one or more general purpose computers, specialized server computers (including, by way of example, PC (personal computer) servers, UNIX® servers, mid-range servers, mainframe computers, rack-mounted servers, etc.), server farms, server clusters, or any other appropriate arrangement and/or combination. In various embodiments, server 3012 may be adapted to run one or more services or software applications described in the foregoing disclosure. For example, server 3012 may correspond to a server for performing processing described above according to an embodiment of the present disclosure.

[0150] Server 3012 may run an operating system including any of those discussed above, as well as any commercially available server operating system. Server 3012 may also run any of a variety of additional server applications and/or mid-tier applications, including HTTP (hypertext transport protocol) servers, FTP (file transfer protocol) servers, CGI (common gateway interface)

servers, JAVA® servers, database servers, and the like. Exemplary database servers include without limitation those commercially available from Oracle, Microsoft, Sybase, IBM (International Business Machines), and the like.

[0151] In some implementations, server 3012 may include one or more applications to analyze and consolidate data feeds and/or event updates received from users of client computing devices 3002, 3004, 3006, and 3008. As an example, data feeds and/or event updates may include, but are not limited to, Twitter® feeds, Facebook® updates or real-time updates received from one or more third party information sources and continuous data streams, which may include real-time events related to sensor data applications, financial tickers, network performance measuring tools (e.g., network monitoring and traffic management applications), clickstream analysis tools, automobile traffic monitoring, and the like. Server 3012 may also include one or more applications to display the data feeds and/or real-time events via one or more display devices of client computing devices 3002, 3004, 3006, and 3008.

[0152] Distributed system 3000 may also include one or more databases 3014 and 3016. Databases 3014 and 3016 may reside in a variety of locations. By way of example, one or more of databases 3014 and 3016 may reside on a non-transitory storage medium local to (and/or resident in) server 3012. Alternatively, databases 3014 and 3016 may be remote from server 3012 and in communication with server 3012 via a network-based or dedicated connection. In one set of embodiments, databases 3014 and 3016 may reside in a storage-area network (SAN). Similarly, any necessary files for performing the functions attributed to server 3012 may be stored locally on server 3012 and/or remotely, as appropriate. In one set of embodiments, databases 3014 and 3016 may include relational databases, such as databases provided by Oracle, that are adapted to store, update, and retrieve data in response to SQL-formatted commands.

[0153] **FIG. 31** is a simplified block diagram of one or more components of a system environment 3100 by which services provided by one or more components of an embodiment system may be offered as cloud services, in accordance with an embodiment of the present disclosure. In the illustrated embodiment, system environment 3100 includes one or more client computing devices 3104, 3106, and 3108 that may be used by users to interact with a cloud infrastructure system 3102 that provides cloud services. The client computing devices may be

configured to operate a client application such as a web browser, a proprietary client application (e.g., Oracle Forms), or some other application, which may be used by a user of the client computing device to interact with cloud infrastructure system 3102 to use services provided by cloud infrastructure system 3102.

[0154] It should be appreciated that cloud infrastructure system 3102 depicted in the figure may have other components than those depicted. Further, the embodiment shown in the figure is only one example of a cloud infrastructure system that may incorporate an embodiment of the invention. In some other embodiments, cloud infrastructure system 3102 may have more or fewer components than shown in the figure, may combine two or more components, or may have a different configuration or arrangement of components.

[0155] Client computing devices 3104, 3106, and 3108 may be devices similar to those described above for 3002, 3004, 3006, and 3008.

[0156] Although exemplary system environment 3100 is shown with three client computing devices, any number of client computing devices may be supported. Other devices such as devices with sensors, etc. may interact with cloud infrastructure system 3102.

[0157] Network(s) 3110 may facilitate communications and exchange of data between clients 3104, 3106, and 3108 and cloud infrastructure system 3102. Each network may be any type of network familiar to those skilled in the art that can support data communications using any of a variety of commercially-available protocols, including those described above for network(s) 3010.

[0158] Cloud infrastructure system 3102 may comprise one or more computers and/or servers that may include those described above for server 3012.

[0159] In certain embodiments, services provided by the cloud infrastructure system may include a host of services that are made available to users of the cloud infrastructure system on demand, such as online data storage and backup solutions, Web-based e-mail services, hosted office suites and document collaboration services, database processing, managed technical support services, and the like. Services provided by the cloud infrastructure system can dynamically scale to meet the needs of its users. A specific instantiation of a service provided by cloud infrastructure system is referred to herein as a "service instance." In general, any service

made available to a user via a communication network, such as the Internet, from a cloud service provider's system is referred to as a "cloud service." Typically, in a public cloud environment, servers and systems that make up the cloud service provider's system are different from the customer's own on-premises servers and systems. For example, a cloud service provider's system may host an application, and a user may, via a communication network such as the Internet, on demand, order and use the application.

[0160] In some examples, a service in a computer network cloud infrastructure may include protected computer network access to storage, a hosted database, a hosted web server, a software application, or other service provided by a cloud vendor to a user, or as otherwise known in the art. For example, a service can include password-protected access to remote storage on the cloud through the Internet. As another example, a service can include a web service-based hosted relational database and a script-language middleware engine for private use by a networked developer. As another example, a service can include access to an email software application hosted on a cloud vendor's web site.

[0161] In certain embodiments, cloud infrastructure system 3102 may include a suite of applications, middleware, and database service offerings that are delivered to a customer in a self-service, subscription-based, elastically scalable, reliable, highly available, and secure manner. An example of such a cloud infrastructure system is the Oracle Public Cloud provided by the present assignee.

[0162] In various embodiments, cloud infrastructure system 3102 may be adapted to automatically provision, manage and track a customer's subscription to services offered by cloud infrastructure system 3102. Cloud infrastructure system 3102 may provide the cloud services via different deployment models. For example, services may be provided under a public cloud model in which cloud infrastructure system 3102 is owned by an organization selling cloud services (e.g., owned by Oracle) and the services are made available to the general public or different industry enterprises. As another example, services may be provided under a private cloud model in which cloud infrastructure system 3102 is operated solely for a single organization and may provide services for one or more entities within the organization. The cloud services may also be provided under a community cloud model in which cloud infrastructure system 3102 and the services provided by cloud infrastructure system 3102 are

shared by several organizations in a related community. The cloud services may also be provided under a hybrid cloud model, which is a combination of two or more different models.

[0163] In some embodiments, the services provided by cloud infrastructure system 3102 may include one or more services provided under Software as a Service (SaaS) category, Platform as a Service (PaaS) category, Infrastructure as a Service (IaaS) category, or other categories of services including hybrid services. A customer, via a subscription order, may order one or more services provided by cloud infrastructure system 3102. Cloud infrastructure system 3102 then performs processing to provide the services in the customer's subscription order.

[0164] In some embodiments, the services provided by cloud infrastructure system 3102 may include, without limitation, application services, platform services and infrastructure services. In some examples, application services may be provided by the cloud infrastructure system via a SaaS platform. The SaaS platform may be configured to provide cloud services that fall under the SaaS category. For example, the SaaS platform may provide capabilities to build and deliver a suite of on-demand applications on an integrated development and deployment platform. The SaaS platform may manage and control the underlying software and infrastructure for providing the SaaS services. By utilizing the services provided by the SaaS platform, customers can utilize applications executing on the cloud infrastructure system. Customers can acquire the application services without the need for customers to purchase separate licenses and support. Various different SaaS services may be provided. Examples include, without limitation, services that provide solutions for sales performance management, enterprise integration, and business flexibility for large organizations.

[0165] In some embodiments, platform services may be provided by the cloud infrastructure system via a PaaS platform. The PaaS platform may be configured to provide cloud services that fall under the PaaS category. Examples of platform services may include without limitation services that enable organizations (such as Oracle) to consolidate existing applications on a shared, common architecture, as well as the ability to build new applications that leverage the shared services provided by the platform. The PaaS platform may manage and control the underlying software and infrastructure for providing the PaaS services. Customers can acquire the PaaS services provided by the cloud infrastructure system without the need for customers to

purchase separate licenses and support. Examples of platform services include, without limitation, Oracle Java Cloud Service (JCS), Oracle Database Cloud Service (DBCS), and others.

[0166] By utilizing the services provided by the PaaS platform, customers can employ programming languages and tools supported by the cloud infrastructure system and also control the deployed services. In some embodiments, platform services provided by the cloud infrastructure system may include database cloud services, middleware cloud services (e.g., Oracle Fusion Middleware services), and Java cloud services. In one embodiment, database cloud services may support shared service deployment models that enable organizations to pool database resources and offer customers a Database as a Service in the form of a database cloud. Middleware cloud services may provide a platform for customers to develop and deploy various business applications, and Java cloud services may provide a platform for customers to deploy Java applications, in the cloud infrastructure system.

[0167] Various different infrastructure services may be provided by an IaaS platform in the cloud infrastructure system. The infrastructure services facilitate the management and control of the underlying computing resources, such as storage, networks, and other fundamental computing resources for customers utilizing services provided by the SaaS platform and the PaaS platform.

[0168] In certain embodiments, cloud infrastructure system 3102 may also include infrastructure resources 3130 for providing the resources used to provide various services to customers of the cloud infrastructure system. In one embodiment, infrastructure resources 3130 may include pre-integrated and optimized combinations of hardware, such as servers, storage, and networking resources to execute the services provided by the PaaS platform and the SaaS platform.

[0169] In some embodiments, resources in cloud infrastructure system 3102 may be shared by multiple users and dynamically re-allocated per demand. Additionally, resources may be allocated to users in different time zones. For example, cloud infrastructure system 3130 may enable a first set of users in a first time zone to utilize resources of the cloud infrastructure system for a specified number of hours and then enable the re-allocation of the same resources to another set of users located in a different time zone, thereby maximizing the utilization of resources.

[0170] In certain embodiments, a number of internal shared services 3132 may be provided that are shared by different components or modules of cloud infrastructure system 3102 and by the services provided by cloud infrastructure system 3102. These internal shared services may include, without limitation, a security and identity service, an integration service, an enterprise repository service, an enterprise manager service, a virus scanning and white list service, a high availability, backup and recovery service, service for enabling cloud support, an email service, a notification service, a file transfer service, and the like.

[0171] In certain embodiments, cloud infrastructure system 3102 may provide comprehensive management of cloud services (e.g., SaaS, PaaS, and IaaS services) in the cloud infrastructure system. In one embodiment, cloud management functionality may include capabilities for provisioning, managing and tracking a customer's subscription received by cloud infrastructure system 3102, and the like.

[0172] In one embodiment, as depicted in the figure, cloud management functionality may be provided by one or more modules, such as an order management module 3120, an order orchestration module 3122, an order provisioning module 3124, an order management and monitoring module 3126, and an identity management module 3128. These modules may include or be provided using one or more computers and/or servers, which may be general purpose computers, specialized server computers, server farms, server clusters, or any other appropriate arrangement and/or combination.

[0173] In exemplary operation 3134, a customer using a client device, such as client device 3104, 3106 or 3108, may interact with cloud infrastructure system 3102 by requesting one or more services provided by cloud infrastructure system 3102 and placing an order for a subscription for one or more services offered by cloud infrastructure system 3102. In certain embodiments, the customer may access a cloud User Interface (UI), cloud UI 3112, cloud UI 3114 and/or cloud UI 3116 and place a subscription order via these UIs. The order information received by cloud infrastructure system 3102 in response to the customer placing an order may include information identifying the customer and one or more services offered by the cloud infrastructure system 3102 that the customer intends to subscribe to.

[0174] After an order has been placed by the customer, the order information is received via the cloud UIs, 3112, 3114 and/or 3116.

[0175] At operation 3136, the order is stored in order database 3118. Order database 3118 can be one of several databases operated by cloud infrastructure system 3118 and operated in conjunction with other system elements.

[0176] At operation 3138, the order information is forwarded to an order management module 3120. In some instances, order management module 3120 may be configured to perform billing and accounting functions related to the order, such as verifying the order, and upon verification, booking the order.

[0177] At operation 3140, information regarding the order is communicated to an order orchestration module 3122. Order orchestration module 3122 may utilize the order information to orchestrate the provisioning of services and resources for the order placed by the customer. In some instances, order orchestration module 3122 may orchestrate the provisioning of resources to support the subscribed services using the services of order provisioning module 3124.

[0178] In certain embodiments, order orchestration module 3122 enables the management of business processes associated with each order and applies business logic to determine whether an order should proceed to provisioning. At operation 3142, upon receiving an order for a new subscription, order orchestration module 3122 sends a request to order provisioning module 3124 to allocate resources and configure those resources needed to fulfill the subscription order. Order provisioning module 3124 enables the allocation of resources for the services ordered by the customer. Order provisioning module 3124 provides a level of abstraction between the cloud services provided by cloud infrastructure system 3100 and the physical implementation layer that is used to provision the resources for providing the requested services. Order orchestration module 3122 may thus be isolated from implementation details, such as whether or not services and resources are actually provisioned on the fly or pre-provisioned and only allocated/assigned upon request.

[0179] At operation 3144, once the services and resources are provisioned, a notification of the provided service may be sent to customers on client devices 3104, 3106 and/or 3108 by order provisioning module 3124 of cloud infrastructure system 3102.

[0180] At operation 3146, the customer's subscription order may be managed and tracked by an order management and monitoring module 3126. In some instances, order management and monitoring module 3126 may be configured to collect usage statistics for the services in the

subscription order, such as the amount of storage used, the amount data transferred, the number of users, and the amount of system up time and system down time.

[0181] In certain embodiments, cloud infrastructure system 3100 may include an identity management module 3128. Identity management module 3128 may be configured to provide identity services, such as access management and authorization services in cloud infrastructure system 3100. In some embodiments, identity management module 3128 may control information about customers who wish to utilize the services provided by cloud infrastructure system 3102. Such information can include information that authenticates the identities of such customers and information that describes which actions those customers are authorized to perform relative to various system resources (e.g., files, directories, applications, communication ports, memory segments, etc.) Identity management module 3128 may also include the management of descriptive information about each customer and about how and by whom that descriptive information can be accessed and modified.

[0182] **FIG. 32** illustrates an exemplary computer system 3200, in which various embodiments of the present invention may be implemented. The system 3200 may be used to implement any of the computer systems described above. As shown in the figure, computer system 3200 includes a processing unit 3204 that communicates with a number of peripheral subsystems via a bus subsystem 3202. These peripheral subsystems may include a processing acceleration unit 3206, an I/O subsystem 3208, a storage subsystem 3218 and a communications subsystem 3224. Storage subsystem 3218 includes tangible computer-readable storage media 3222 and a system memory 3210.

[0183] Bus subsystem 3202 provides a mechanism for letting the various components and subsystems of computer system 3200 communicate with each other as intended. Although bus subsystem 3202 is shown schematically as a single bus, alternative embodiments of the bus subsystem may utilize multiple buses. Bus subsystem 3202 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. For example, such architectures may include an Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral

Component Interconnect (PCI) bus, which can be implemented as a Mezzanine bus manufactured to the IEEE P1386.1 standard.

[0184] Processing unit 3204, which can be implemented as one or more integrated circuits (e.g., a conventional microprocessor or microcontroller), controls the operation of computer system 3200. One or more processors may be included in processing unit 3204. These processors may include single core or multicore processors. In certain embodiments, processing unit 3204 may be implemented as one or more independent processing units 3232 and/or 3234 with single or multicore processors included in each processing unit. In other embodiments, processing unit 3204 may also be implemented as a quad-core processing unit formed by integrating two dual-core processors into a single chip.

[0185] In various embodiments, processing unit 3204 can execute a variety of programs in response to program code and can maintain multiple concurrently executing programs or processes. At any given time, some or all of the program code to be executed can be resident in processor(s) 3204 and/or in storage subsystem 3218. Through suitable programming, processor(s) 3204 can provide various functionalities described above. Computer system 3200 may additionally include a processing acceleration unit 3206, which can include a digital signal processor (DSP), a special-purpose processor, and/or the like.

[0186] I/O subsystem 3208 may include user interface input devices and user interface output devices. User interface input devices may include a keyboard, pointing devices such as a mouse or trackball, a touchpad or touch screen incorporated into a display, a scroll wheel, a click wheel, a dial, a button, a switch, a keypad, audio input devices with voice command recognition systems, microphones, and other types of input devices. User interface input devices may include, for example, motion sensing and/or gesture recognition devices such as the Microsoft Kinect® motion sensor that enables users to control and interact with an input device, such as the Microsoft Xbox® 360 game controller, through a natural user interface using gestures and spoken commands. User interface input devices may also include eye gesture recognition devices such as the Google Glass® blink detector that detects eye activity (e.g., 'blinking' while taking pictures and/or making a menu selection) from users and transforms the eye gestures as input into an input device (e.g., Google Glass®). Additionally, user interface input devices may

include voice recognition sensing devices that enable users to interact with voice recognition systems (e.g., Siri® navigator), through voice commands.

[0187] User interface input devices may also include, without limitation, three dimensional (3D) mice, joysticks or pointing sticks, gamepads and graphic tablets, and audio/visual devices such as speakers, digital cameras, digital camcorders, portable media players, webcams, image scanners, fingerprint scanners, barcode reader 3D scanners, 3D printers, laser rangefinders, and eye gaze tracking devices. Additionally, user interface input devices may include, for example, medical imaging input devices such as computed tomography, magnetic resonance imaging, position emission tomography, medical ultrasonography devices. User interface input devices may also include, for example, audio input devices such as MIDI keyboards, digital musical instruments and the like.

[0188] User interface output devices may include a display subsystem, indicator lights, or non-visual displays such as audio output devices, etc. The display subsystem may be a cathode ray tube (CRT), a flat-panel device, such as that using a liquid crystal display (LCD) or plasma display, a projection device, a touch screen, and the like. In general, use of the term "output device" is intended to include all possible types of devices and mechanisms for outputting information from computer system 3200 to a user or other computer. For example, user interface output devices may include, without limitation, a variety of display devices that visually convey text, graphics and audio/video information such as monitors, printers, speakers, headphones, automotive navigation systems, plotters, voice output devices, and modems.

[0189] Computer system 3200 may comprise a storage subsystem 3218 that comprises software elements, shown as being currently located within a system memory 3210. System memory 3210 may store program instructions that are loadable and executable on processing unit 3204, as well as data generated during the execution of these programs.

[0190] Depending on the configuration and type of computer system 3200, system memory 3210 may be volatile (such as random access memory (RAM)) and/or non-volatile (such as read-only memory (ROM), flash memory, etc.) The RAM typically contains data and/or program modules that are immediately accessible to and/or presently being operated and executed by processing unit 3204. In some implementations, system memory 3210 may include multiple different types of memory, such as static random access memory (SRAM) or dynamic random

access memory (DRAM). In some implementations, a basic input/output system (BIOS), containing the basic routines that help to transfer information between elements within computer system 3200, such as during start-up, may typically be stored in the ROM. By way of example, and not limitation, system memory 3210 also illustrates application programs 3212, which may include client applications, Web browsers, mid-tier applications, relational database management systems (RDBMS), etc., program data 3214, and an operating system 3216. By way of example, operating system 3216 may include various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems, a variety of commercially-available UNIX® or UNIX-like operating systems (including without limitation the variety of GNU/Linux operating systems, the Google Chrome® OS, and the like) and/or mobile operating systems such as iOS, Windows® Phone, Android® OS, BlackBerry® 10 OS, and Palm® OS operating systems.

[0191] Storage subsystem 3218 may also provide a tangible computer-readable storage medium for storing the basic programming and data constructs that provide the functionality of some embodiments. Software (programs, code modules, instructions) that when executed by a processor provide the functionality described above may be stored in storage subsystem 3218. These software modules or instructions may be executed by processing unit 3204. Storage subsystem 3218 may also provide a repository for storing data used in accordance with the present invention.

[0192] Storage subsystem 3200 may also include a computer-readable storage media reader 3220 that can further be connected to computer-readable storage media 3222. Together and, optionally, in combination with system memory 3210, computer-readable storage media 3222 may comprehensively represent remote, local, fixed, and/or removable storage devices plus storage media for temporarily and/or more permanently containing, storing, transmitting, and retrieving computer-readable information.

[0193] Computer-readable storage media 3222 containing code, or portions of code, can also include any appropriate media known or used in the art, including storage media and communication media, such as but not limited to, volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage and/or transmission of information. This can include tangible computer-readable storage media such as RAM, ROM, electronically erasable programmable ROM (EEPROM), flash memory or other memory

technology, CD-ROM, digital versatile disk (DVD), or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or other tangible computer readable media. This can also include nontangible computer-readable media, such as data signals, data transmissions, or any other medium which can be used to transmit the desired information and which can be accessed by computing system 3200.

[0194] By way of example, computer-readable storage media 3222 may include a hard disk drive that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive that reads from or writes to a removable, nonvolatile magnetic disk, and an optical disk drive that reads from or writes to a removable, nonvolatile optical disk such as a CD ROM, DVD, and Blu-Ray® disk, or other optical media. Computer-readable storage media 3222 may include, but is not limited to, Zip® drives, flash memory cards, universal serial bus (USB) flash drives, secure digital (SD) cards, DVD disks, digital video tape, and the like. Computer-readable storage media 3222 may also include, solid-state drives (SSD) based on non-volatile memory such as flash-memory based SSDs, enterprise flash drives, solid state ROM, and the like, SSDs based on volatile memory such as solid state RAM, dynamic RAM, static RAM, DRAM-based SSDs, magnetoresistive RAM (MRAM) SSDs, and hybrid SSDs that use a combination of DRAM and flash memory based SSDs. The disk drives and their associated computer-readable media may provide non-volatile storage of computer-readable instructions, data structures, program modules, and other data for computer system 3200.

[0195] Communications subsystem 3224 provides an interface to other computer systems and networks. Communications subsystem 3224 serves as an interface for receiving data from and transmitting data to other systems from computer system 3200. For example, communications subsystem 3224 may enable computer system 3200 to connect to one or more devices via the Internet. In some embodiments communications subsystem 3224 can include radio frequency (RF) transceiver components for accessing wireless voice and/or data networks (e.g., using cellular telephone technology, advanced data network technology, such as 3G, 4G or EDGE (enhanced data rates for global evolution), WiFi (IEEE 802.11 family standards, or other mobile communication technologies, or any combination thereof), global positioning system (GPS) receiver components, and/or other components. In some embodiments communications subsystem 3224 can provide wired network connectivity (e.g., Ethernet) in addition to or instead of a wireless interface.

[0196] In some embodiments, communications subsystem 3224 may also receive input communication in the form of structured and/or unstructured data feeds 3226, event streams 3228, event updates 3230, and the like on behalf of one or more users who may use computer system 3200.

[0197] By way of example, communications subsystem 3224 may be configured to receive data feeds 3226 in real-time from users of social networks and/or other communication services such as Twitter® feeds, Facebook® updates, web feeds such as Rich Site Summary (RSS) feeds, and/or real-time updates from one or more third party information sources.

[0198] Additionally, communications subsystem 3224 may also be configured to receive data in the form of continuous data streams, which may include event streams 3228 of real-time events and/or event updates 3230, that may be continuous or unbounded in nature with no explicit end. Examples of applications that generate continuous data may include, for example, sensor data applications, financial tickers, network performance measuring tools (e.g. network monitoring and traffic management applications), clickstream analysis tools, automobile traffic monitoring, and the like.

[0199] Communications subsystem 3224 may also be configured to output the structured and/or unstructured data feeds 3226, event streams 3228, event updates 3230, and the like to one or more databases that may be in communication with one or more streaming data source computers coupled to computer system 3200.

[0200] Computer system 3200 can be one of various types, including a handheld portable device (e.g., an iPhone® cellular phone, an iPad® computing tablet, a PDA), a wearable device (e.g., a Google Glass® head mounted display), a PC, a workstation, a mainframe, a kiosk, a server rack, or any other data processing system.

[0201] Due to the ever-changing nature of computers and networks, the description of computer system 3200 depicted in the figure is intended only as a specific example. Many other configurations having more or fewer components than the system depicted in the figure are possible. For example, customized hardware might also be used and/or particular elements might be implemented in hardware, firmware, software (including applets), or a combination. Further, connection to other computing devices, such as network input/output devices, may be

employed. Based on the disclosure and teachings provided herein, a person of ordinary skill in the art will appreciate other ways and/or methods to implement the various embodiments.

[0202] In the foregoing description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of various embodiments of the present invention. It will be apparent, however, to one skilled in the art that embodiments of the present invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

[0203] The foregoing description provides exemplary embodiments only, and is not intended to limit the scope, applicability, or configuration of the disclosure. Rather, the foregoing description of the exemplary embodiments will provide those skilled in the art with an enabling description for implementing an exemplary embodiment. It should be understood that various changes may be made in the function and arrangement of elements without departing from the spirit and scope of the invention as set forth in the appended claims.

[0204] Specific details are given in the foregoing description to provide a thorough understanding of the embodiments. However, it will be understood by one of ordinary skill in the art that the embodiments may be practiced without these specific details. For example, circuits, systems, networks, processes, and other components may have been shown as components in block diagram form in order not to obscure the embodiments in unnecessary detail. In other instances, well-known circuits, processes, algorithms, structures, and techniques may have been shown without unnecessary detail in order to avoid obscuring the embodiments.

[0205] Also, it is noted that individual embodiments may have beeen described as a process which is depicted as a flowchart, a flow diagram, a data flow diagram, a structure diagram, or a block diagram. Although a flowchart may have described the operations as a sequential process, many of the operations can be performed in parallel or concurrently. In addition, the order of the operations may be re-arranged. A process is terminated when its operations are completed, but could have additional steps not included in a figure. A process may correspond to a method, a function, a procedure, a subroutine, a subprogram, etc. When a process corresponds to a function, its termination can correspond to a return of the function to the calling function or the main function.

[0206] The term "computer-readable medium" includes, but is not limited to portable or fixed storage devices, optical storage devices, wireless channels and various other mediums capable of storing, containing, or carrying instruction(s) and/or data. A code segment or machine-executable instructions may represent a procedure, a function, a subprogram, a program, a routine, a subroutine, a module, a software package, a class, or any combination of instructions, data structures, or program statements. A code segment may be coupled to another code segment or a hardware circuit by passing and/or receiving information, data, arguments, parameters, or memory contents. Information, arguments, parameters, data, etc., may be passed, forwarded, or transmitted via any suitable means including memory sharing, message passing, token passing, network transmission, etc.

[0207] Furthermore, embodiments may be implemented by hardware, software, firmware, middleware, microcode, hardware description languages, or any combination thereof. When implemented in software, firmware, middleware or microcode, the program code or code segments to perform the necessary tasks may be stored in a machine readable medium. A processor(s) may perform the necessary tasks.

[0208] In the foregoing specification, aspects of the invention are described with reference to specific embodiments thereof, but those skilled in the art will recognize that the invention is not limited thereto. Various features and aspects of the above-described invention may be used individually or jointly. Further, embodiments can be utilized in any number of environments and applications beyond those described herein without departing from the broader spirit and scope of the specification. The specification and drawings are, accordingly, to be regarded as illustrative rather than restrictive.

[0209] Additionally, for the purposes of illustration, methods were described in a particular order. It should be appreciated that in alternate embodiments, the methods may be performed in a different order than that described. It should also be appreciated that the methods described above may be performed by hardware components or may be embodied in sequences of machine-executable instructions, which may be used to cause a machine, such as a general-purpose or special-purpose processor or logic circuits programmed with the instructions to perform the methods. These machine-executable instructions may be stored on one or more machine readable mediums, such as CD-ROMs or other type of optical disks, floppy diskettes,

ROMs, RAMs, EPROMs, EEPROMs, magnetic or optical cards, flash memory, or other types of machine-readable mediums suitable for storing electronic instructions. Alternatively, the methods may be performed by a combination of hardware and software.

WHAT IS CLAIMED IS:

1. A method of providing runtime debugging for containerized services in container environments, the method comprising:

receiving a request for a service at a container environment, wherein:

the container environment comprises a service mesh and a plurality of services encapsulated in a plurality of containers; and

the service is encapsulated in first one or more containers;

determining that the request should be routed to a debug instance of the service;

instantiating the debug instance of the service, wherein the debug instance is encapsulated in second one or more containers and comprises:

code implementing the service; and

one or more debugging utilities;

routing, by the service mesh, the request to the debug instance.

2. The method of claim 1, wherein the first one or more containers are organized into a container pod.

3. The method of claim 1, wherein the container environment comprises an orchestrated container platform comprising a container scheduler.

4. The method of claim 3, wherein the container scheduler causes the debug instance of the service to be instantiated.

5. The method of claim 1, wherein the container environment comprises an Application Programming Interface (API) registry that causes the debug instance of the service to be instantiated.

6. The method of claim 5, wherein the API registry receives a registration for the debug instance of the service and makes an HTTP endpoint of the debug instance of the service available through an API function call.

7. The method of claim 5, wherein the API registry receives a registration for the service comprising a property indicating that the debug instance of the service should be instantiated.

8. A non-transitory, computer-readable medium comprising instructions that, when executed by one or more processors, causes the one or more processors to perform operations comprising:

receiving a request for a service at a container environment, wherein:

the container environment comprises a service mesh and a plurality of services encapsulated in a plurality of containers; and

the service is encapsulated in first one or more containers;

determining that the request should be routed to a debug instance of the service;

instantiating the debug instance of the service, wherein the debug instance is encapsulated in second one or more containers and comprises:

code implementing the service; and

one or more debugging utilities;

routing, by the service mesh, the request to the debug instance.

9. The non-transitory computer-readable medium according to claim 8 wherein the service is encapsulated in a single container.

10. The non-transitory computer-readable medium according to claim 9 wherein the single container also comprises the one or more debugging utilities.

11. The non-transitory computer-readable medium according to claim 9 wherein the one or more debugging utilities are encapsulated in at least one container other than the single container.

12. The non-transitory computer-readable medium according to claim 9 wherein the one or more debugging utilities comprise a process for monitoring memory usage or processor usage.

13. The non-transitory computer-readable medium according to claim 9 wherein the one or more debugging utilities comprise a debug daemon.

14. The non-transitory computer-readable medium according to claim 9 wherein the code implementing the service comprises a debug build of the service.

15. A system comprising:

one or more processors; and

one or more memory devices comprising instructions that, when executed by the one or more processors, cause the one or more processors to perform operations comprising:

receiving a request for a service at a container environment, wherein:

the container environment comprises a service mesh and a plurality of services encapsulated in a plurality of containers; and

the service is encapsulated in first one or more containers;

determining that the request should be routed to a debug instance of the service;

instantiating the debug instance of the service, wherein the debug instance is encapsulated in second one or more containers and comprises:

code implementing the service; and

one or more debugging utilities;

routing, by the service mesh, the request to the debug instance.

16. The system of claim 15, wherein the debug instance of the service is instantiated prior to receiving the request.

17. The system of claim 15, wherein the debug instance of the service is instantiated in response to receiving the request.

18. The system of claim 15, wherein determining that the request should be routed to the debug instance of the service comprises identifying a source of the request.

19. The system of claim 15, wherein determining that the request should be routed to the debug instance of the service comprises recognizing a header in the request that designates the request as a debug request.

20. The system of claim 15, wherein the request is forwarded to the debug instance of the service without interrupting the routing of other requests to the service.

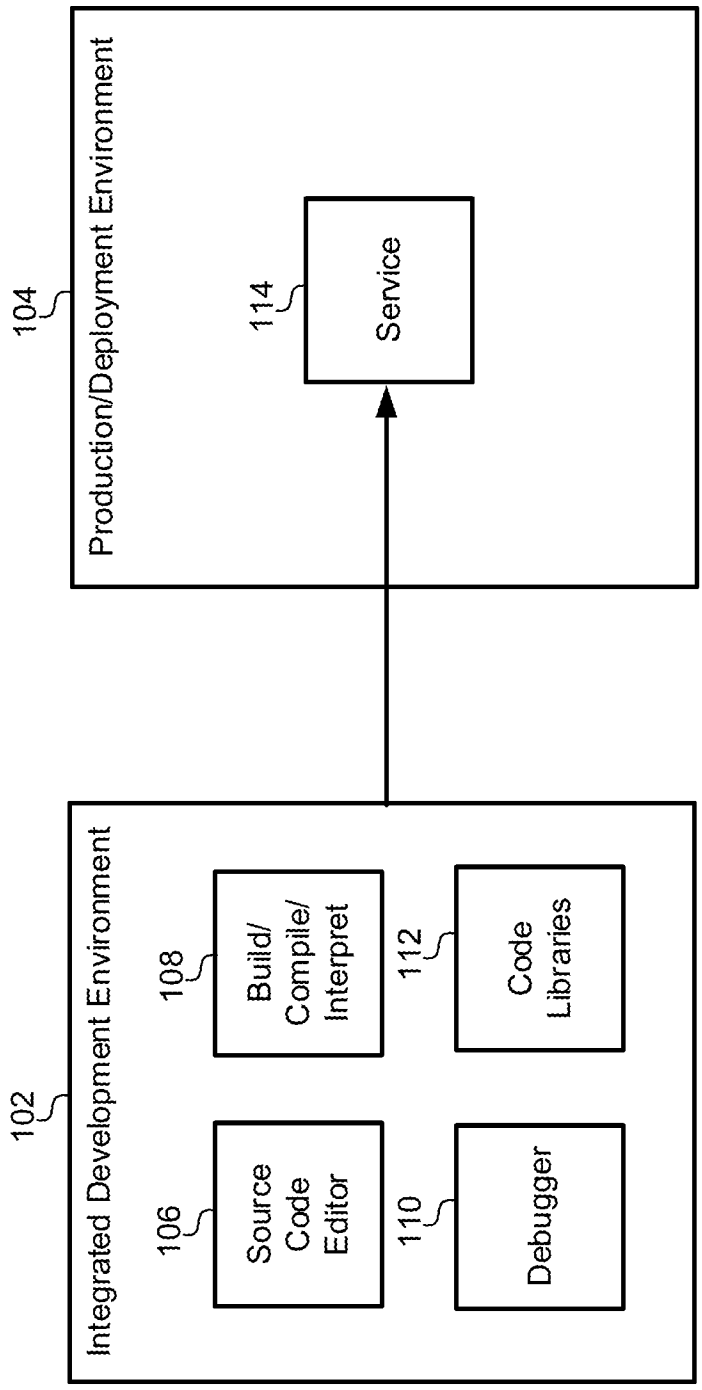
Production/Deployment Environment
104
Service
114
Integrated Development Environment
102
Source Code Editor
106
Build/ Compile/ Interpret
108
Debugger
110
Code Libraries
112

FIG. 1

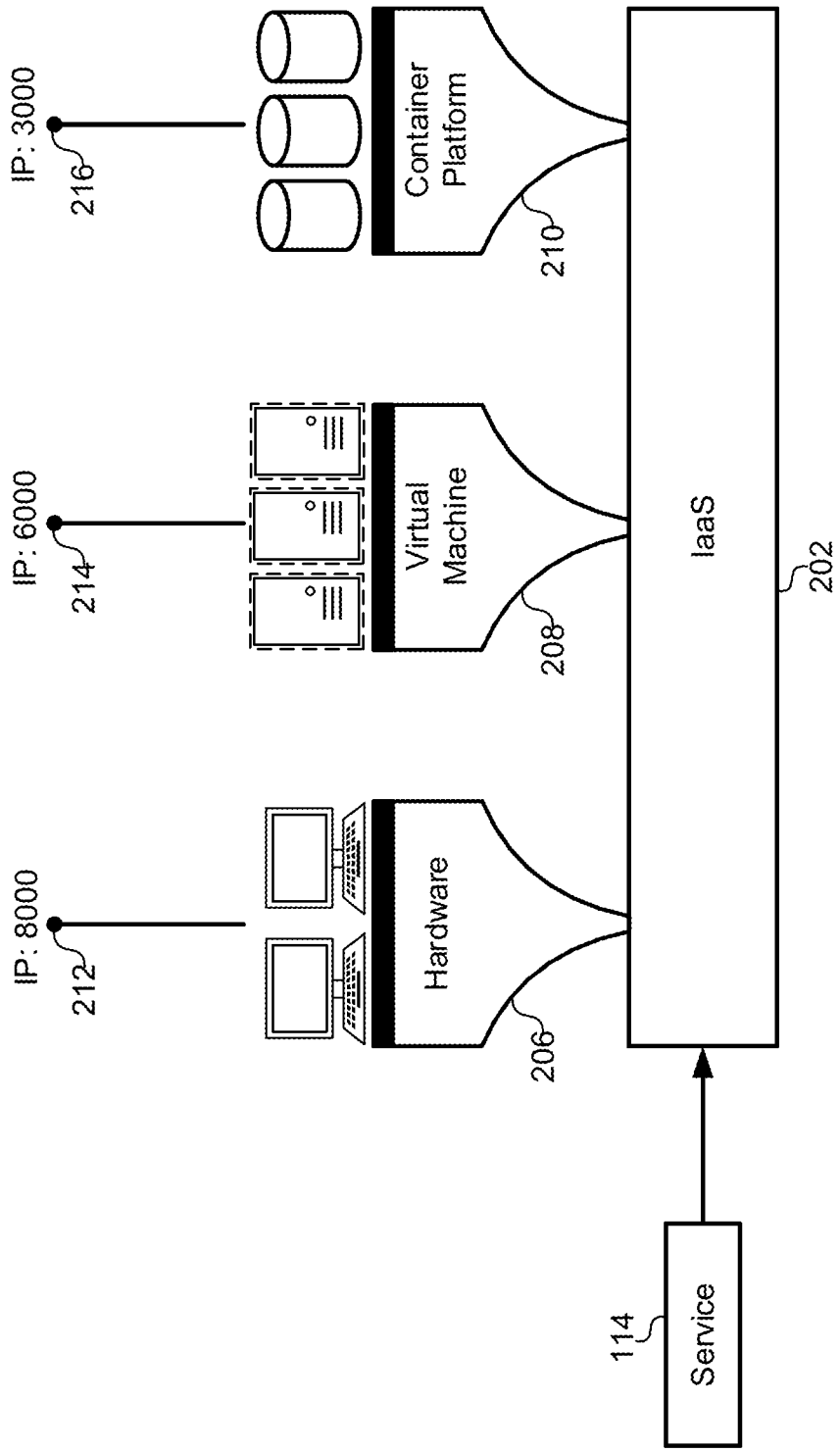
IP: 3000
216
Container Platform
210
IP: 6000
214
Virtual Machine
208
IP: 8000
212
Hardware
206
IaaS
202
114
Service

FIG. 2

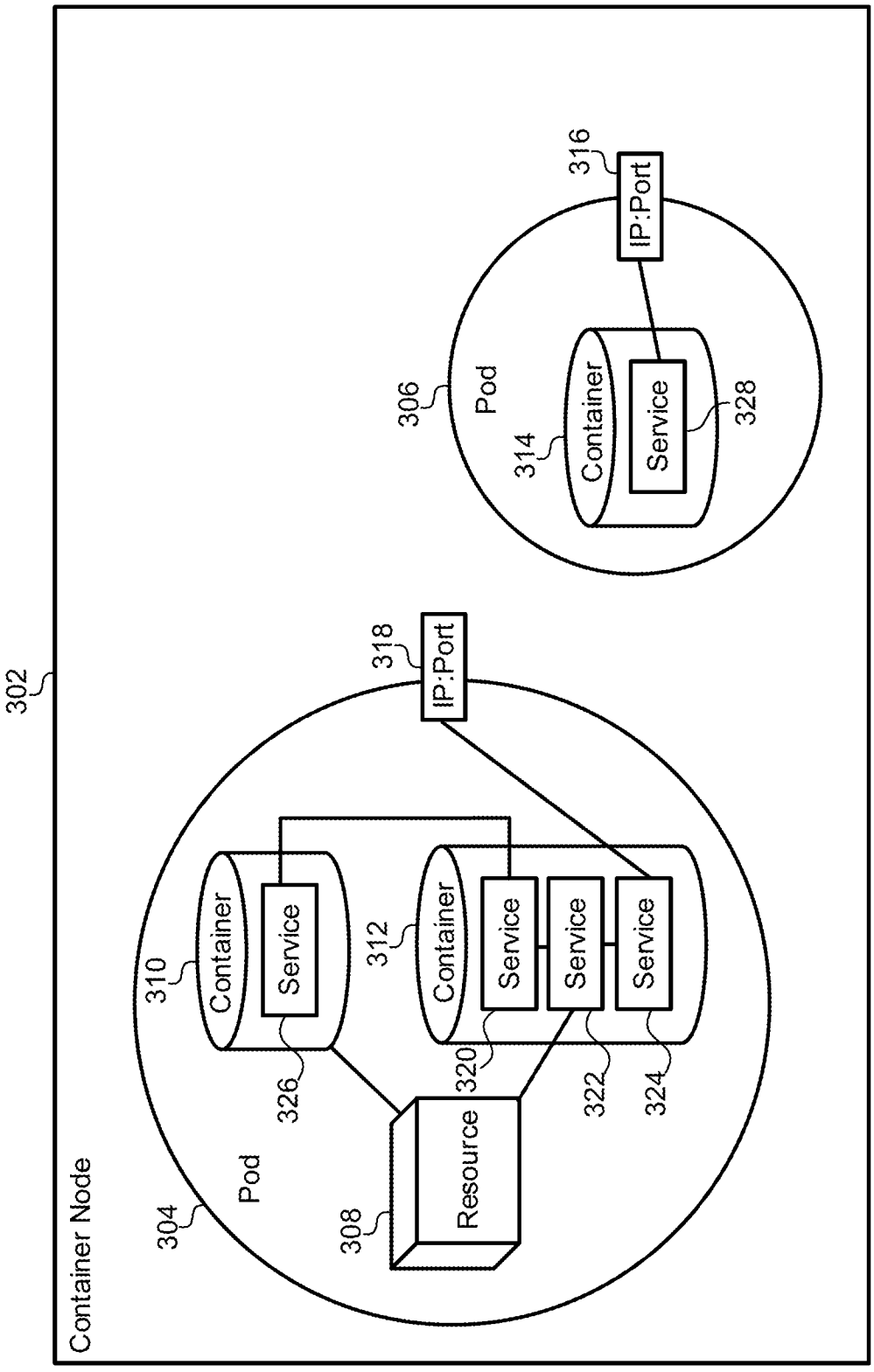
Container Node
302
Pod
304
IP:Port
318
Container
310
Service
326
Container
312
Service
320
Service
322
Service
324
Resource
308
Pod
306
IP:Port
316
Container
314
Service
328

FIG. 3

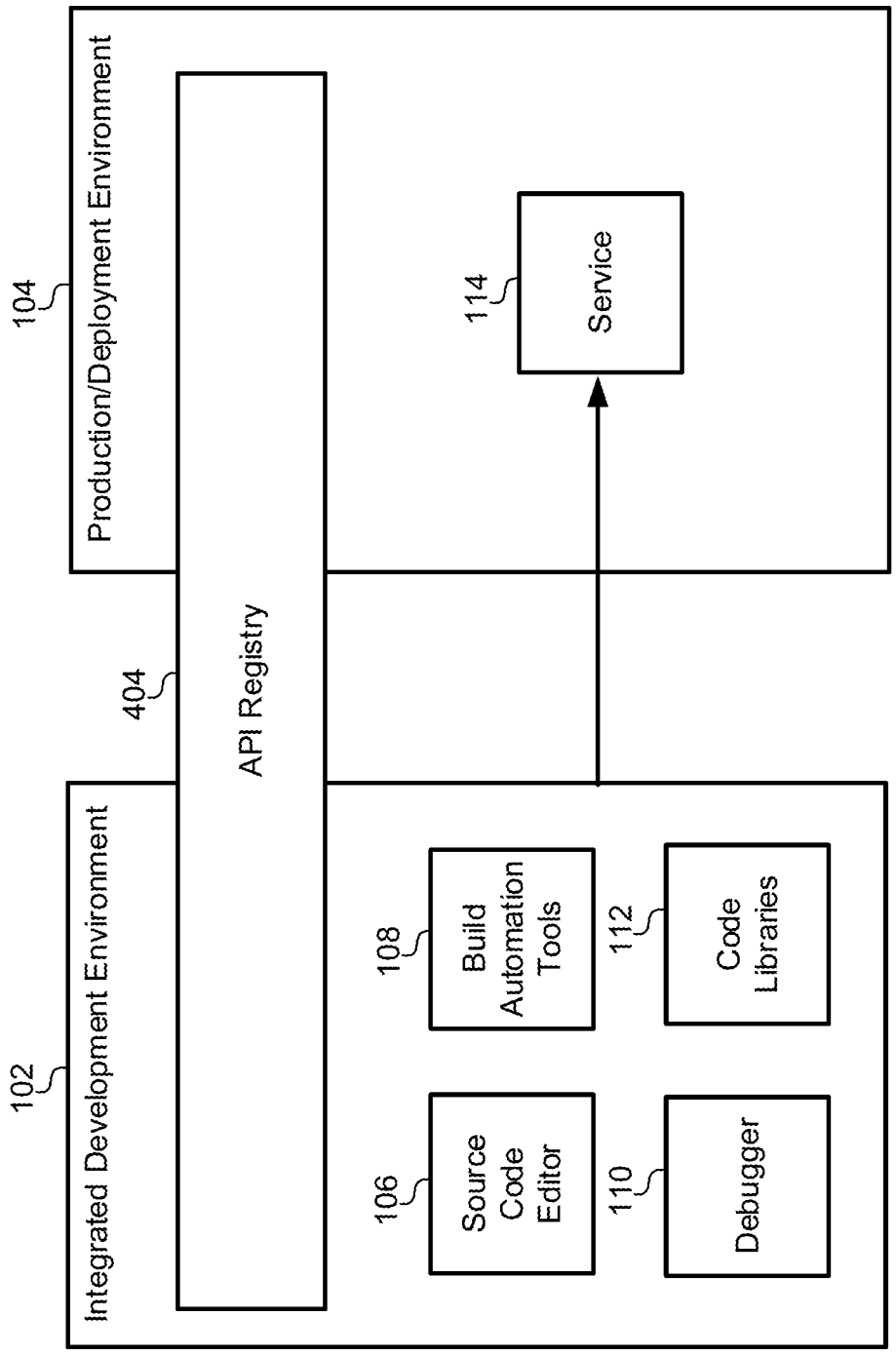
Production/Deployment Environment
104
114
Service
API Registry
404
Integrated Development Environment
102
108
Build Automation Tools
112
Code Libraries
106
Source Code Editor
110
Debugger

FIG. 4

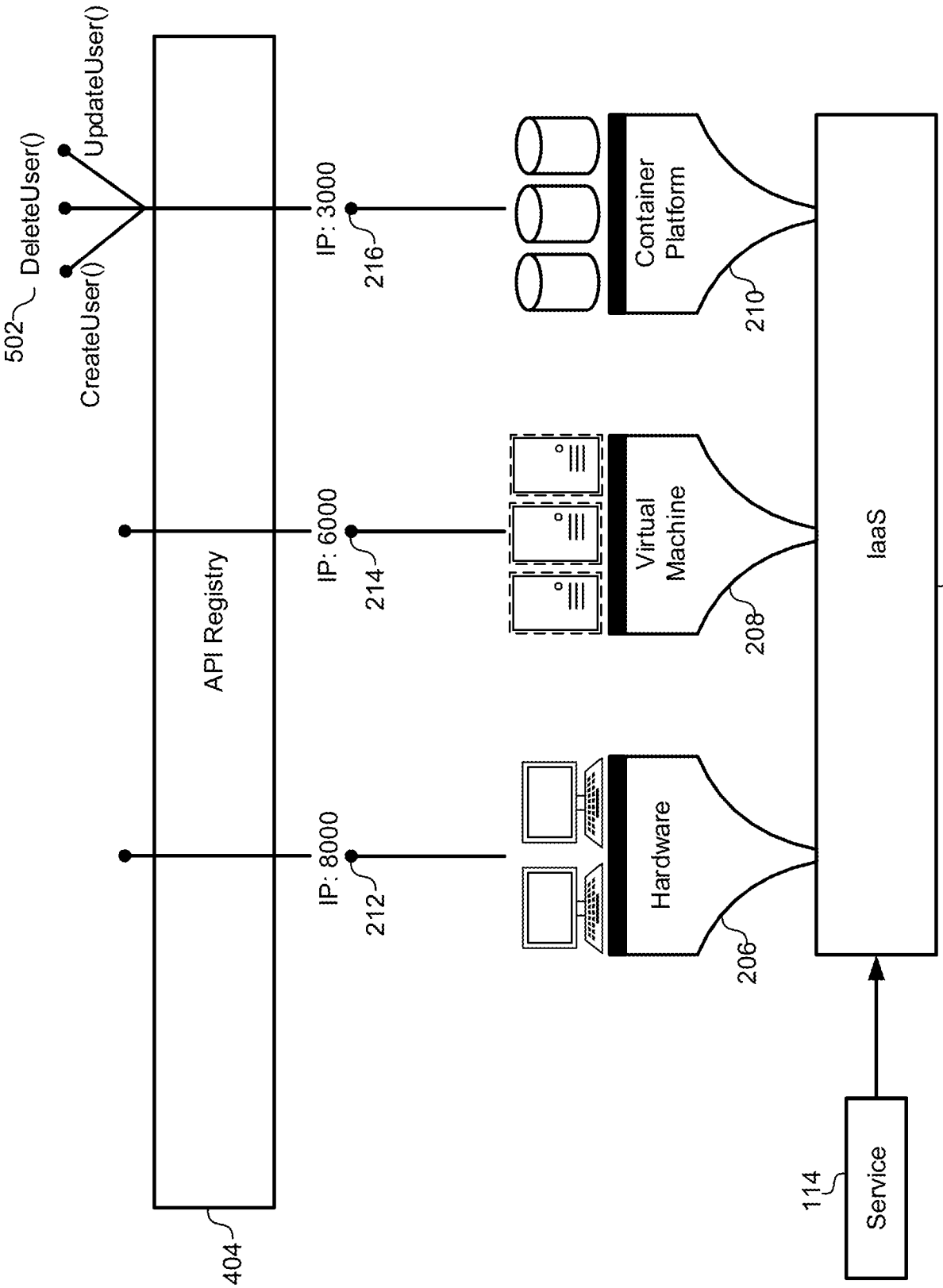
UpdateUser()
DeleteUser()
CreateUser()
502
API Registry
404
IP: 3000
216
IP: 6000
214
IP: 8000
212
Container Platform
210
Virtual Machine
208
Hardware
206
IaaS
202
Service
114

FIG. 5

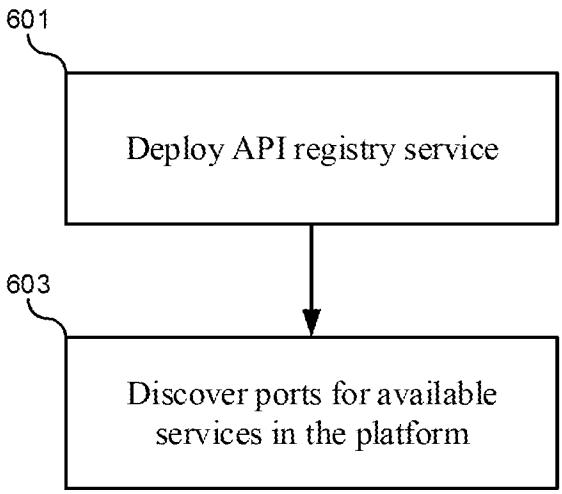
601
Deploy API registry service
603
Discover ports for available services in the platform

FIG. 6A

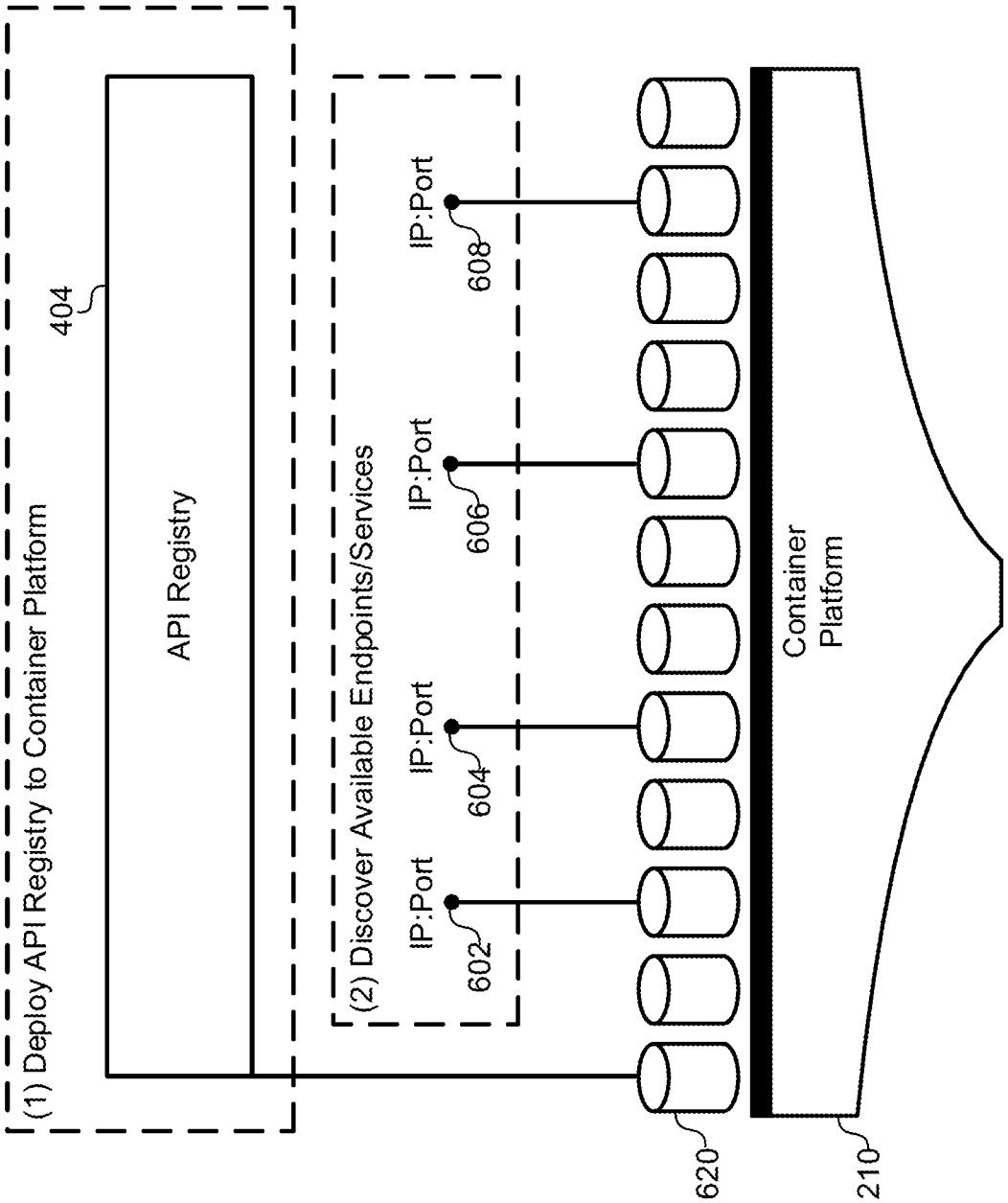
(1) Deploy API Registry to Container Platform
404
API Registry
(2) Discover Available Endpoints/Services
IP:Port
602
IP:Port
604
IP:Port
606
IP:Port
608
620
Container Platform
210

FIG. 6B

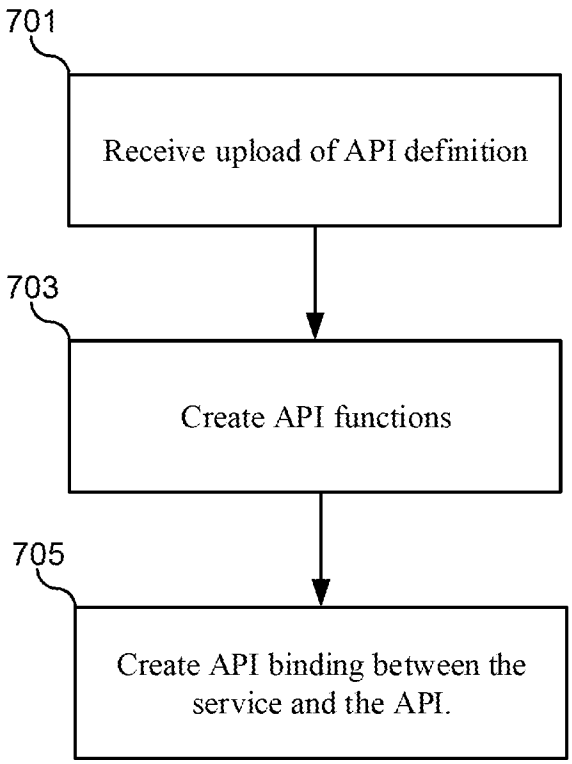
701
Receive upload of API definition
703
Create API functions
705
Create API binding between the service and the API.

FIG. 7A

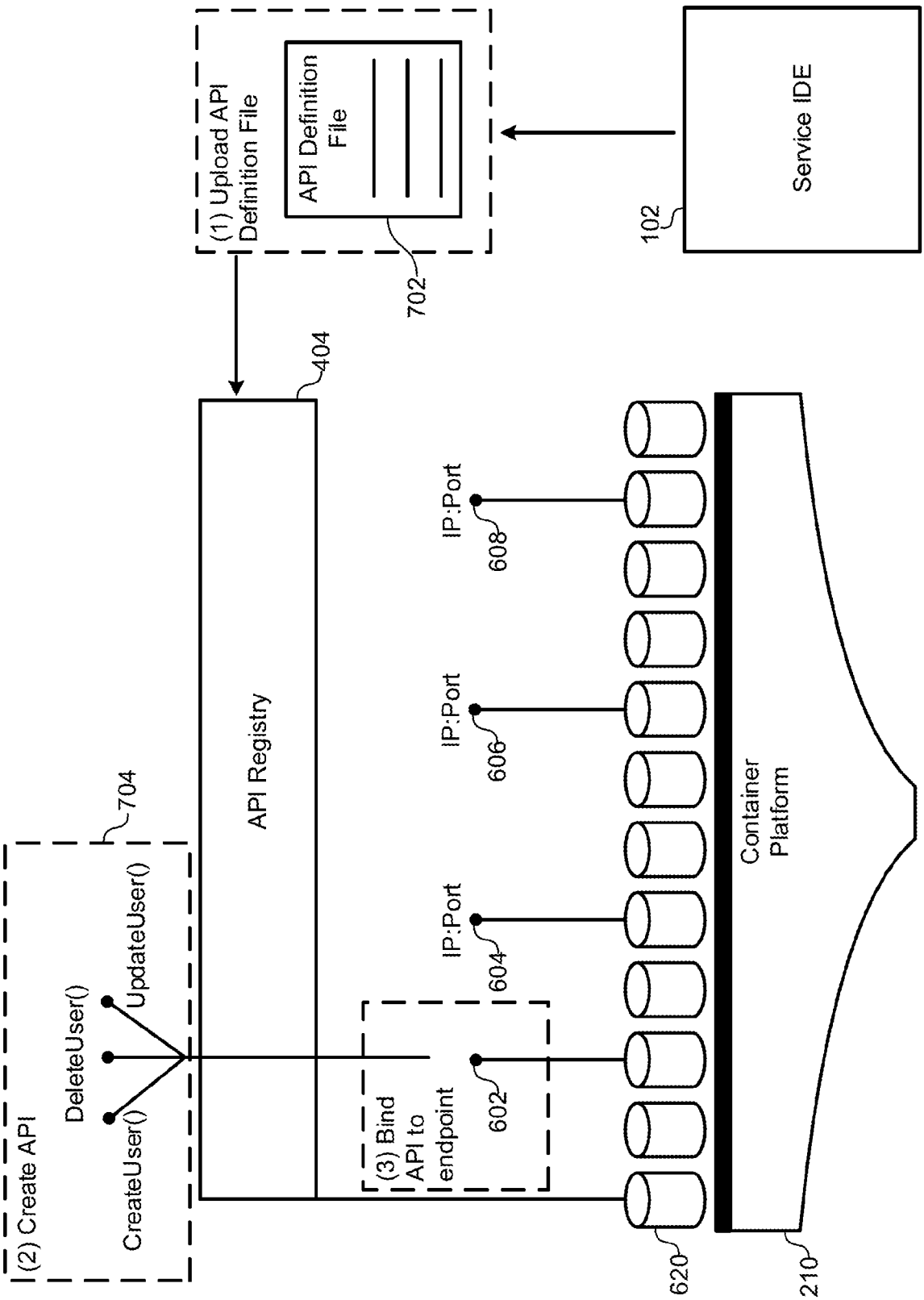
(1) Upload API Definition File
API Definition File
702
Service IDE
102
404
API Registry
704
(2) Create API
CreateUser()
DeleteUser()
UpdateUser()
(3) Bind API to endpoint
602
IP:Port
604
IP:Port
606
IP:Port
608
Container Platform
620
210

FIG. 7B

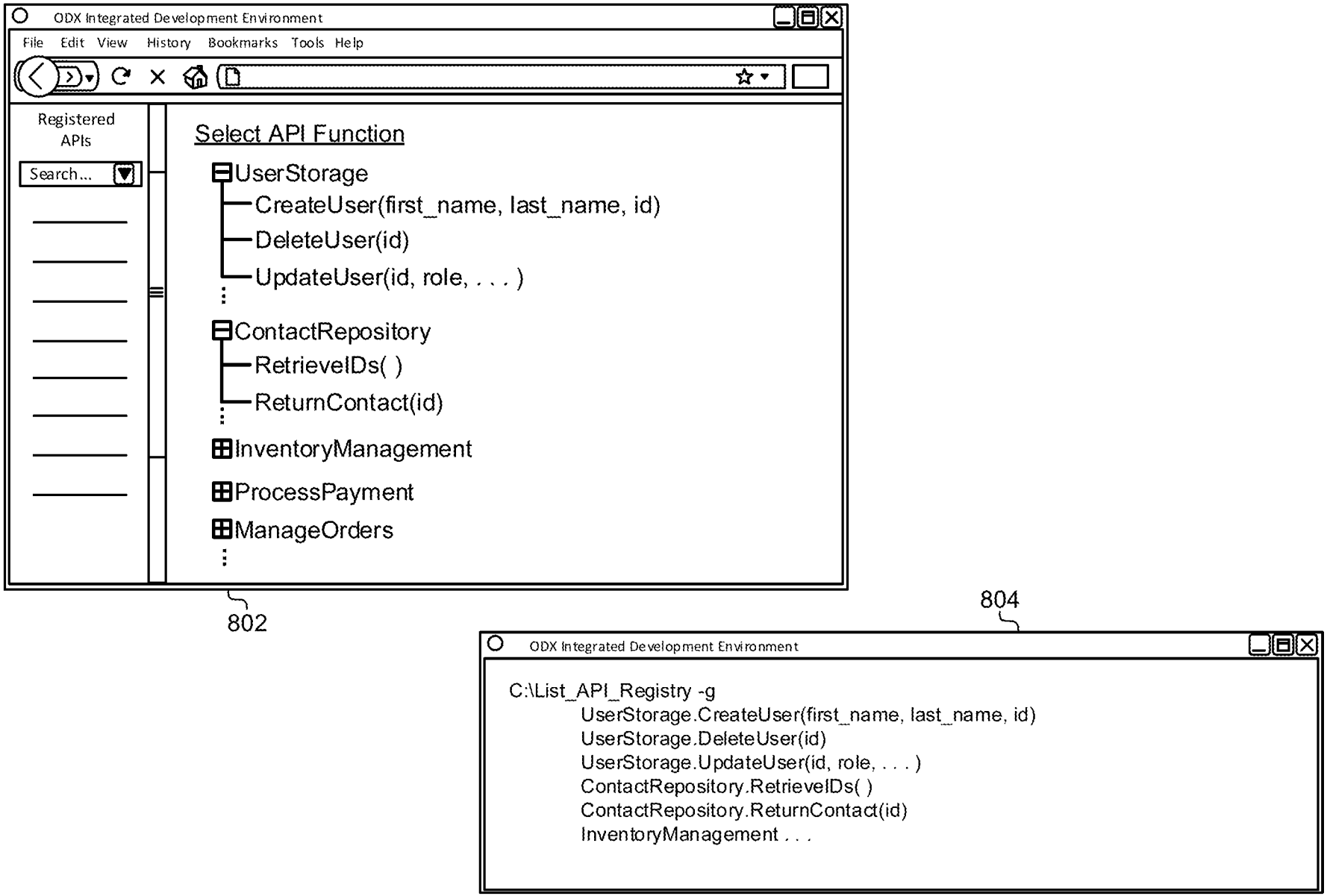
ODX Integrated Development Environment
File Edit View History Bookmarks Tools Help
Registered APIs
Search...
Select API Function
UserStorage
CreateUser(first_name, last_name, id)
DeleteUser(id)
UpdateUser(id, role, . . .)
ContactRepository
RetrieveIDs( )
ReturnContact(id)
InventoryManagement
ProcessPayment
ManageOrders
802
804
ODX Integrated Development Environment
C:\List_API_Registry -g
UserStorage.CreateUser(first_name, last_name, id)
UserStorage.DeleteUser(id)
UserStorage.UpdateUser(id, role, . . .)
ContactRepository.RetrieveIDs( )
ContactRepository.ReturnContact(id)
InventoryManagement . . .

FIG. 8

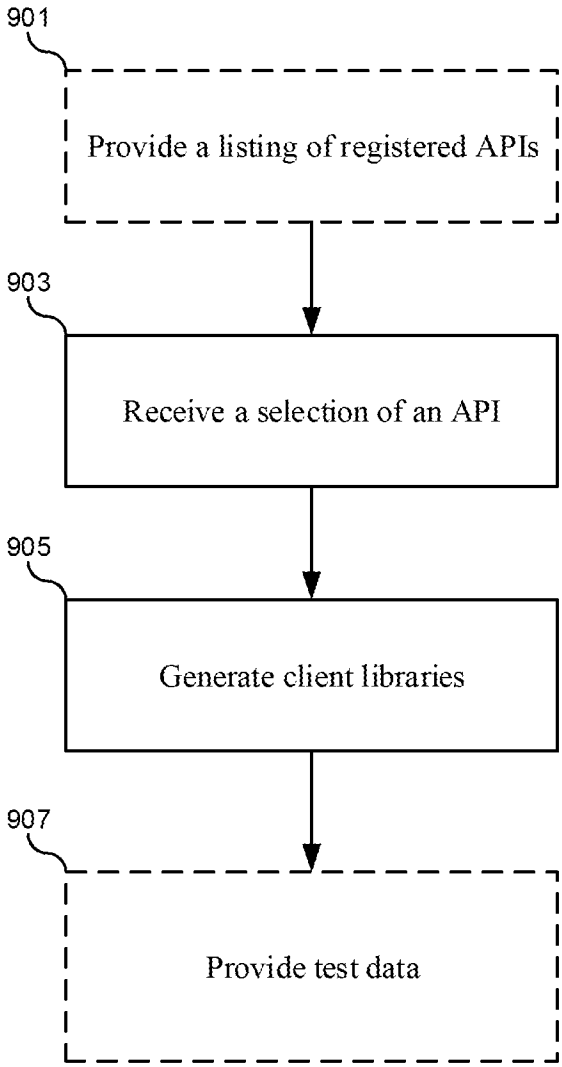
901
Provide a listing of registered APIs
903
Receive a selection of an API
905
Generate client libraries
907
Provide test data

FIG. 9

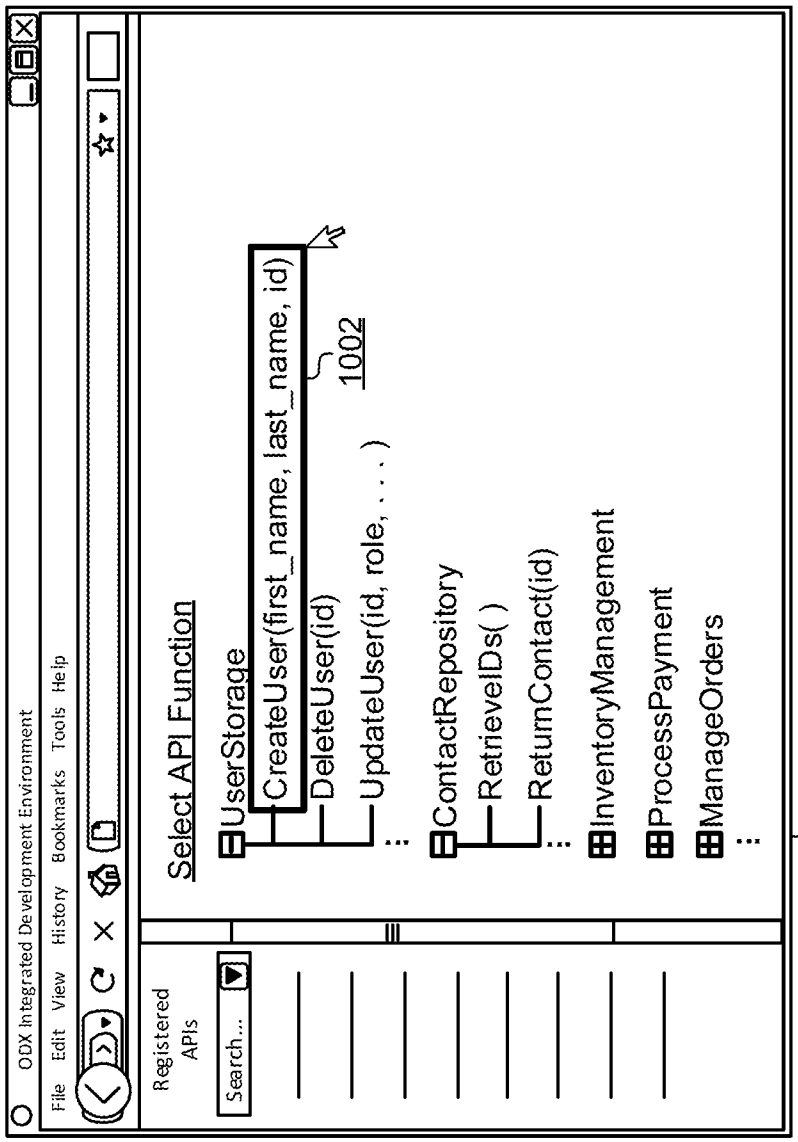
ODX Integrated Development Environment
File Edit View History Bookmarks Tools Help
Registered APIs
Search...
Select API Function
UserStorage
CreateUser(first_name, last_name, id)
DeleteUser(id)
UpdateUser(id, role, . . .)
...
ContactRepository
RetrieveIDs( )
ReturnContact(id)
...
InventoryManagement
ProcessPayment
ManageOrders
...
1002
802

FIG. 10

```
Class User {

Public User CreateUser(str first_name, str last_name, int id) {

        UserName = first_name + last_name

        UserID = id

        Header = . . .

        POST http://192.168.2.100/8000/v1/user/create/<data>

        }

}
```

1102

FIG. 11

```
Class User {

Public User CreateUser(str first_name, str last_name, int id) {

        User.CreateUser(first_name, last_name, id)

        UserName = first_name + last_name

        UserID = id

        Header = . . .

        IPPort = GetIPPort(User.CreateUser)

        POST http://IPPort/v1/user/create/<data>

        }

}
```

1204

1202

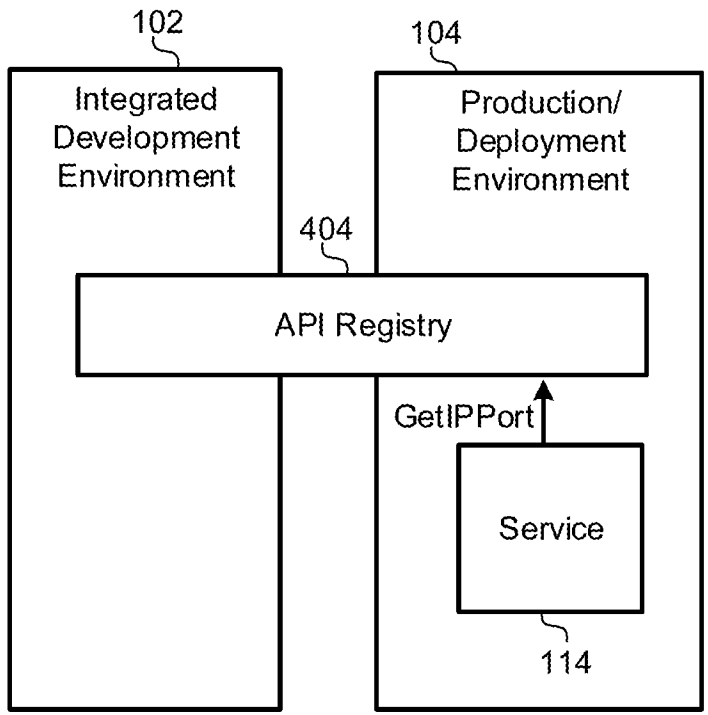

FIG. 12

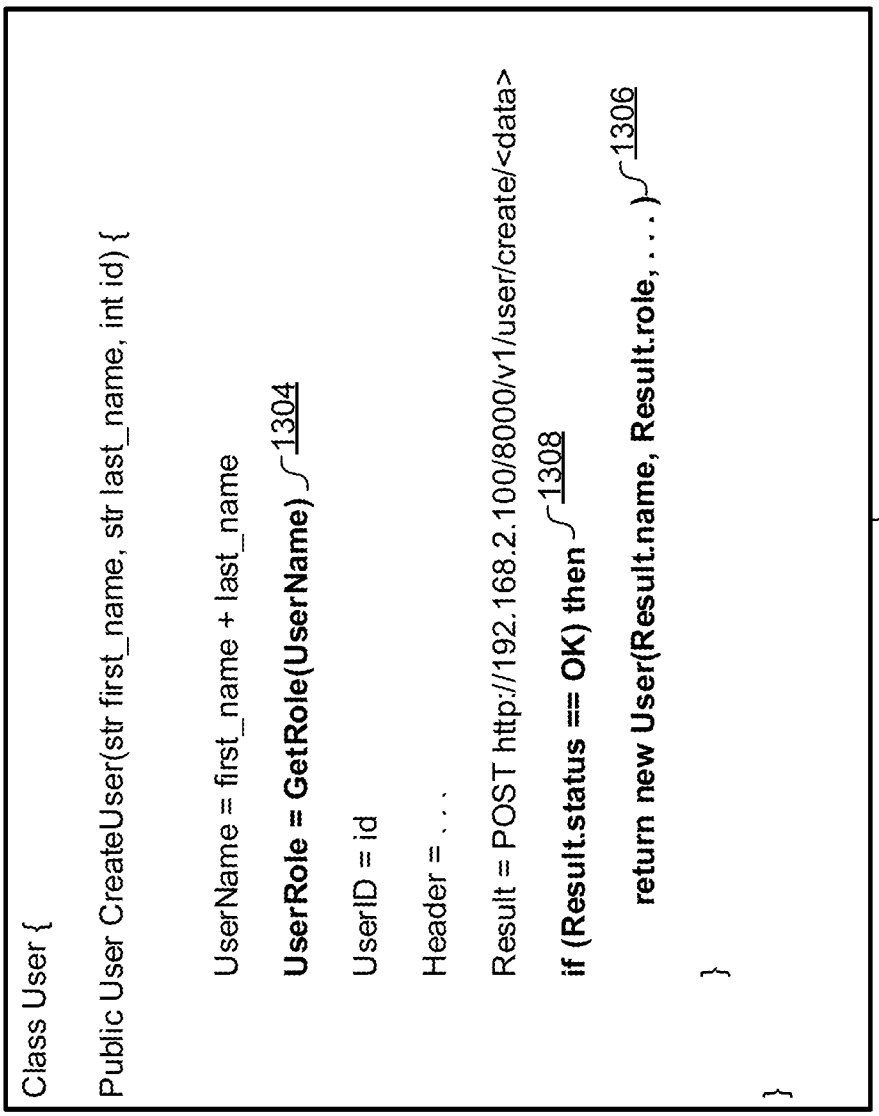
Class User {
Public User CreateUser(str first_name, str last_name, int id) {
UserName = first_name + last_name
UserRole = GetRole(UserName)
1304
UserID = id
Header = . . .
Result = POST http://192.168.2.100/8000/v1/user/create/<data>
if (Result.status == OK) then
1308
return new User(Result.name, Result.role, . . .)
1306
}
}
1302

FIG. 13

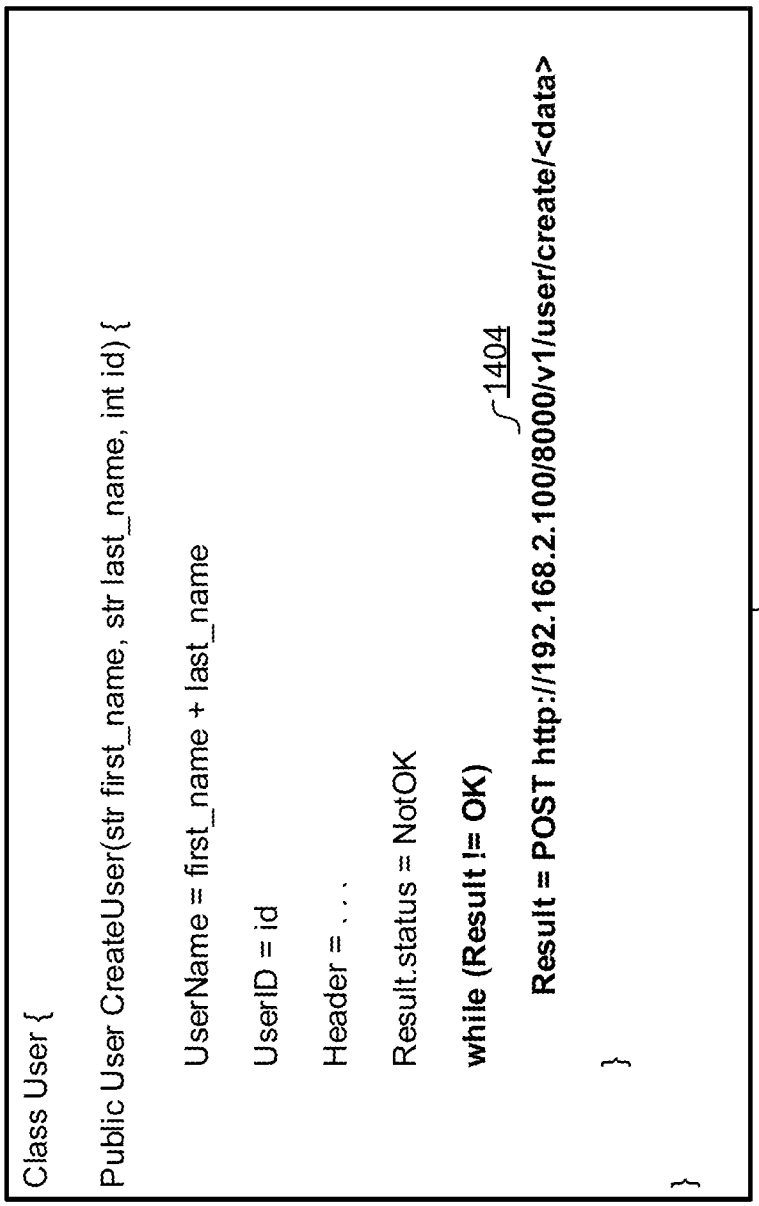
Class User {
Public User CreateUser(str first_name, str last_name, int id) {
UserName = first_name + last_name
UserID = id
Header = . . .
Result.status = NotOK
while (Result != OK)
Result = POST http://192.168.2.100/8000/v1/user/create/<data>
}
}
1404
1402

FIG. 14

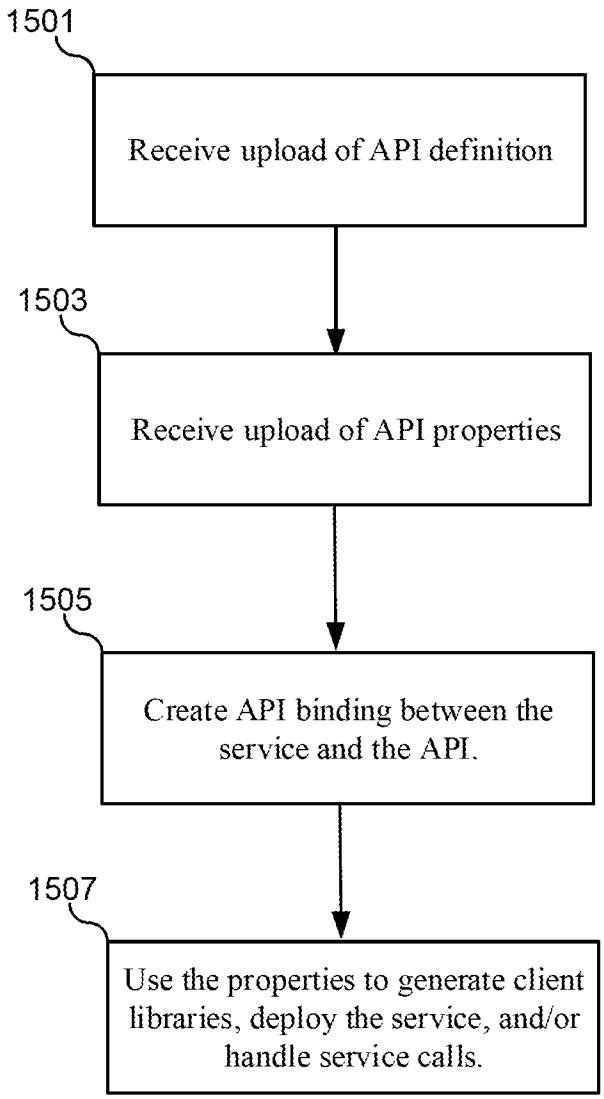
1501
Receive upload of API definition
1503
Receive upload of API properties
1505
Create API binding between the service and the API.
1507
Use the properties to generate client libraries, deploy the service, and/or handle service calls.

FIG. 15A

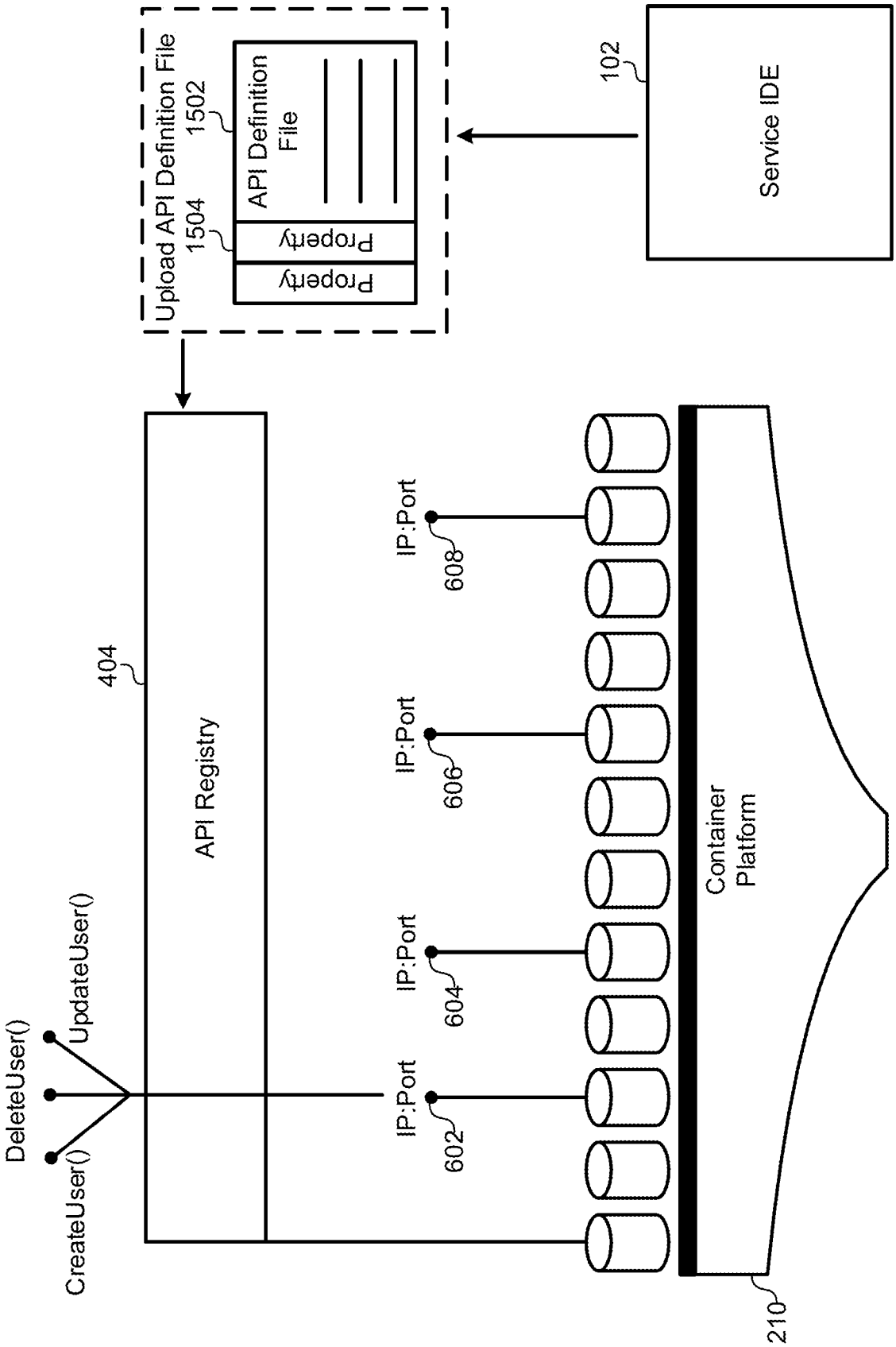
Upload API Definition File
1504
1502
API Definition File
Property
Property
102
Service IDE
404
API Registry
DeleteUser()
UpdateUser()
CreateUser()
IP:Port
602
IP:Port
604
IP:Port
606
IP:Port
608
Container Platform
210

FIG. 15B

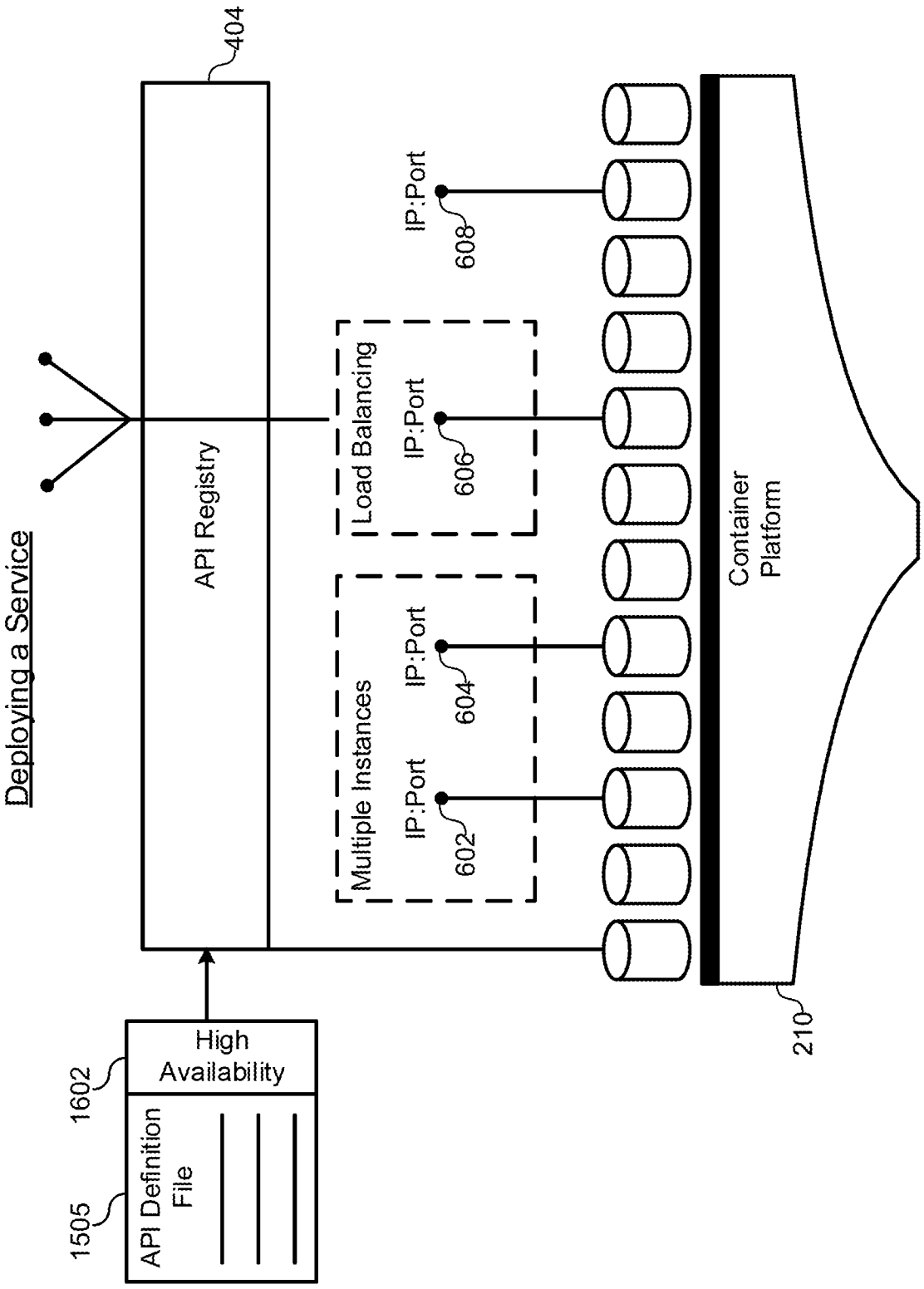
Deploying a Service
404
API Registry
Load Balancing
Multiple Instances
IP:Port
602
IP:Port
604
IP:Port
606
IP:Port
608
Container Platform
210
High Availability
1602
API Definition File
1505

FIG. 16

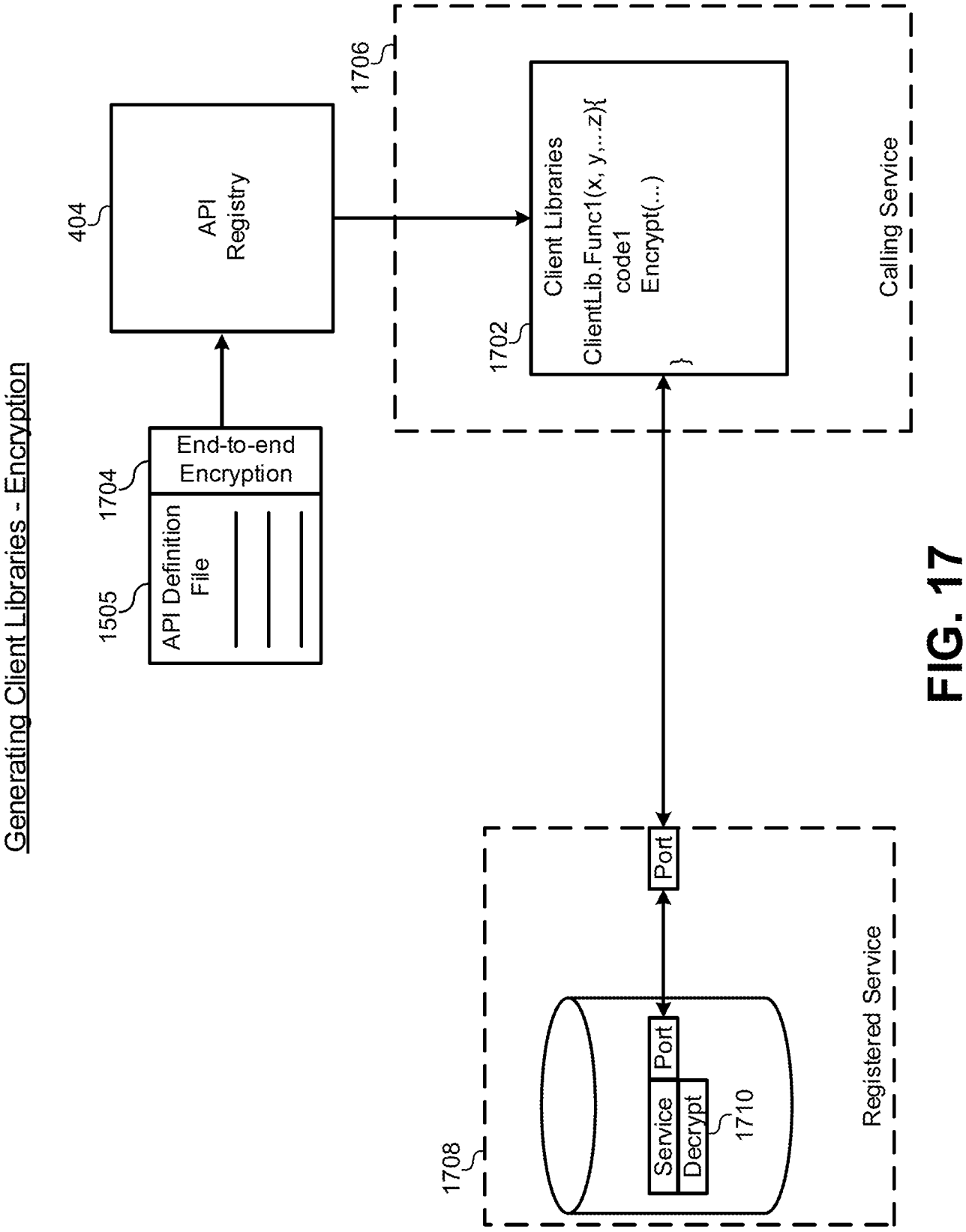
Generating Client Libraries - Encryption
404
API Registry
1706
Client Libraries
ClientLib.Func1(x, y,...z){
code1
Encrypt(...)
}
1702
Calling Service
End-to-end Encryption
1704
API Definition File
1505
Port
Service
Port
Decrypt
1710
1708
Registered Service

FIG. 17

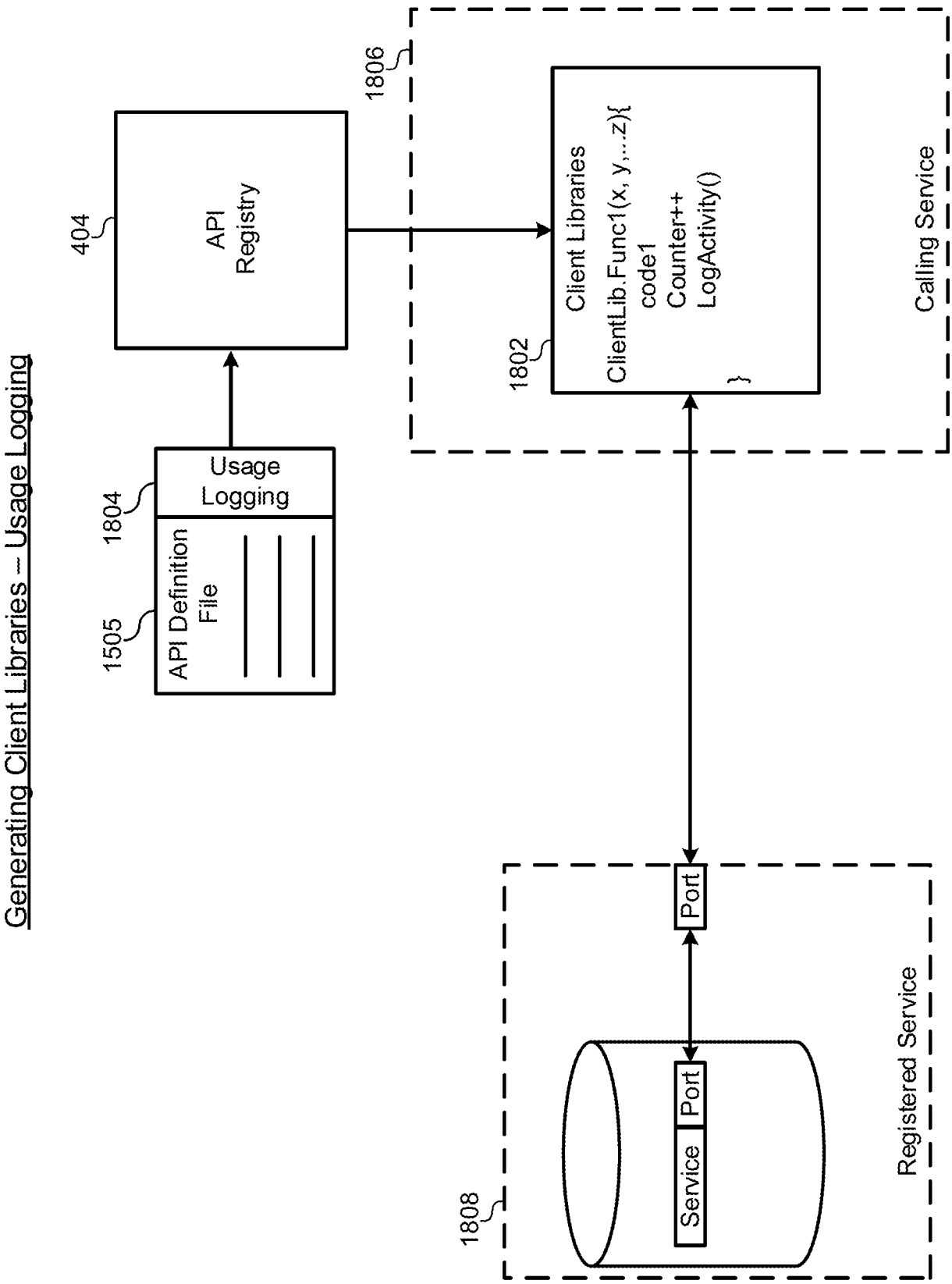
Generating Client Libraries – Usage Logging
404
API Registry
1806
Client Libraries
ClientLib.Func1(x, y,...z){
code1
Counter++
LogActivity()
}
1802
Calling Service
Usage Logging
1804
API Definition File
1505
Port
Service
Port
1808
Registered Service

FIG. 18

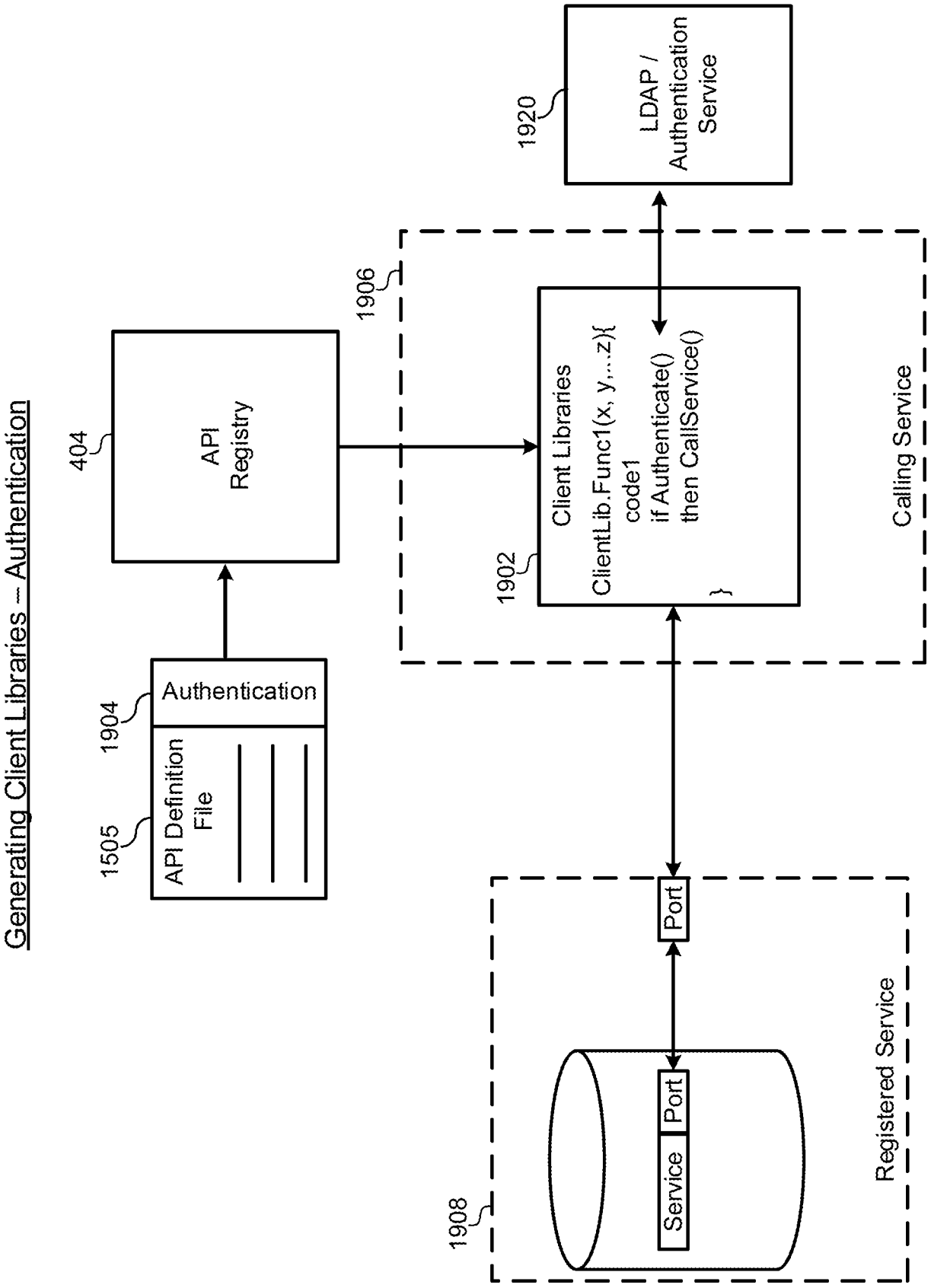
Generating Client Libraries – Authentication
1505
API Definition File
1904
Authentication
404
API Registry
1906
1902
Client Libraries
ClientLib.Func1(x, y,...z){
code1
if Authenticate()
then CallService()
}
Calling Service
1920
LDAP / Authentication Service
Port
Service
Port
1908
Registered Service

FIG. 19

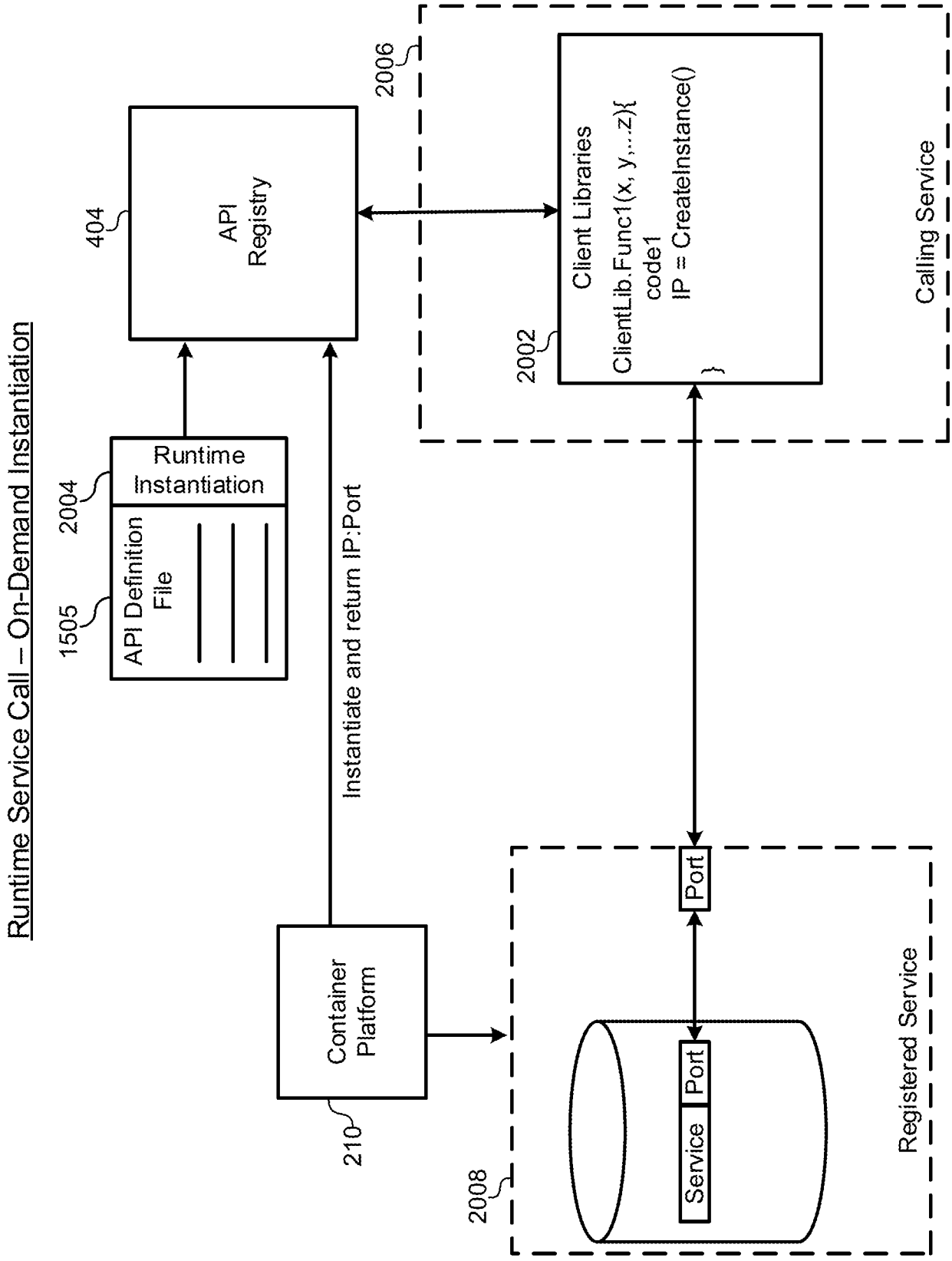
Runtime Service Call – On-Demand Instantiation
2006
Calling Service
Client Libraries
ClientLib.Func1(x, y,...z){
code1
IP = CreateInstance()
}
2002
404
API
Registry
2004
Runtime
Instantiation
1505
API Definition
File
Instantiate and return IP:Port
Container
Platform
210
Port
Service
Port
2008
Registered Service

FIG. 20

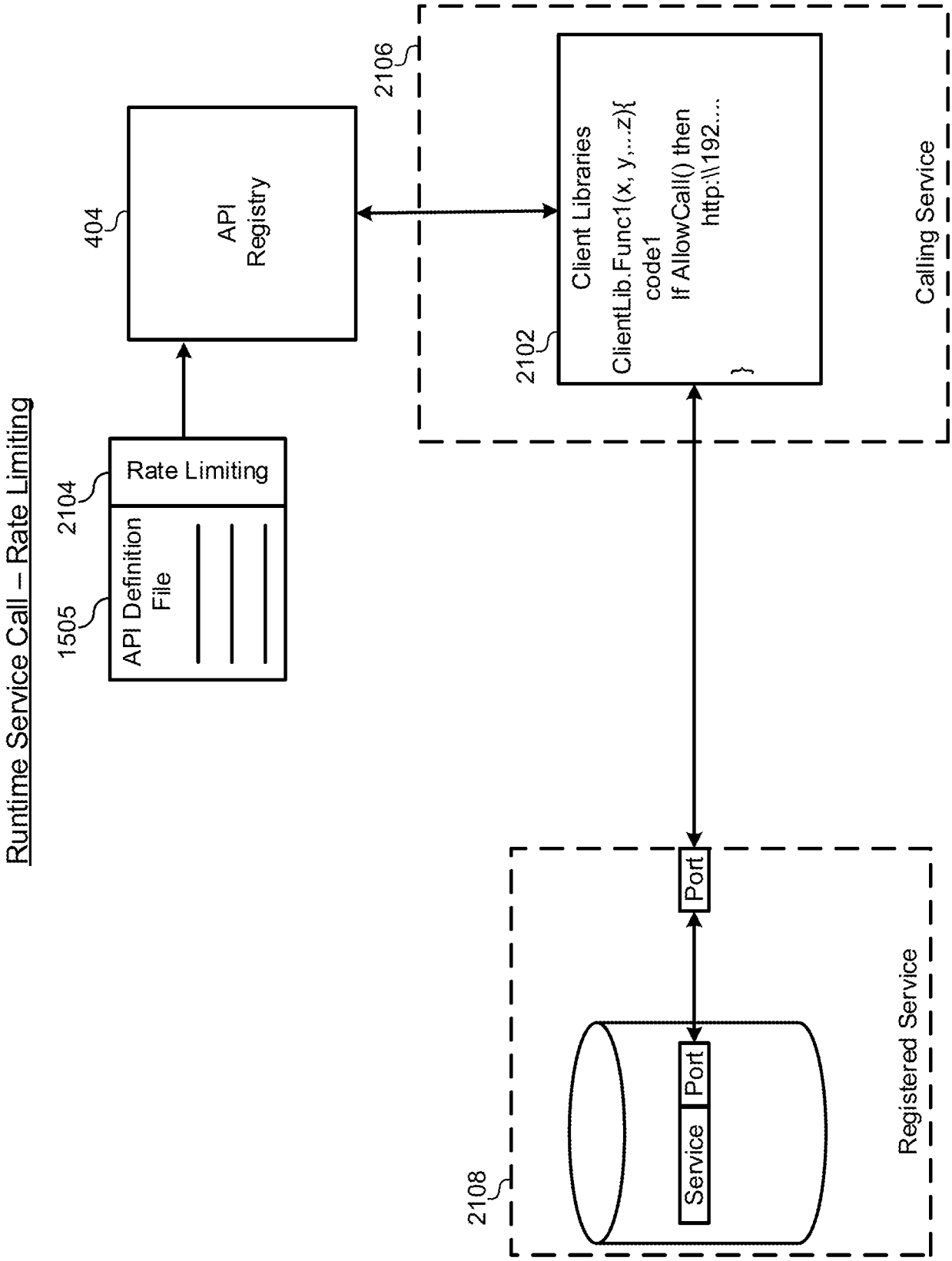
Runtime Service Call – Rate Limiting
API Definition File
1505
Rate Limiting
2104
API Registry
404
2106
Client Libraries
ClientLib.Func1(x, y,...z){
code1
If AllowCall() then
http:\\192....
}
2102
Calling Service
Port
Service
Port
2108
Registered Service

FIG. 21

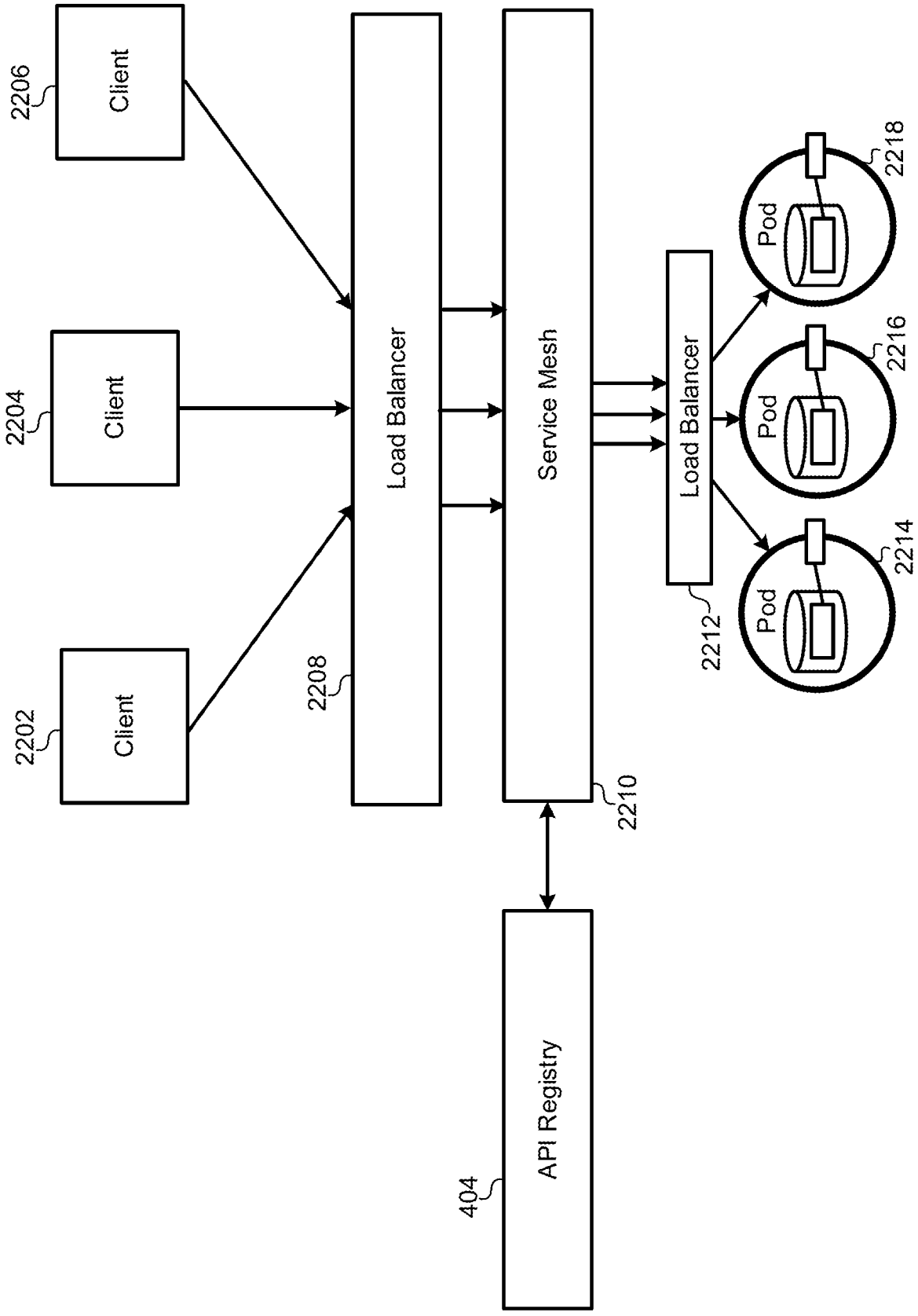
Client
2206
Client
2204
Client
2202
Load Balancer
2208
Service Mesh
2210
Load Balancer
2212
Pod
2218
Pod
2216
Pod
2214
API Registry
404

FIG. 22

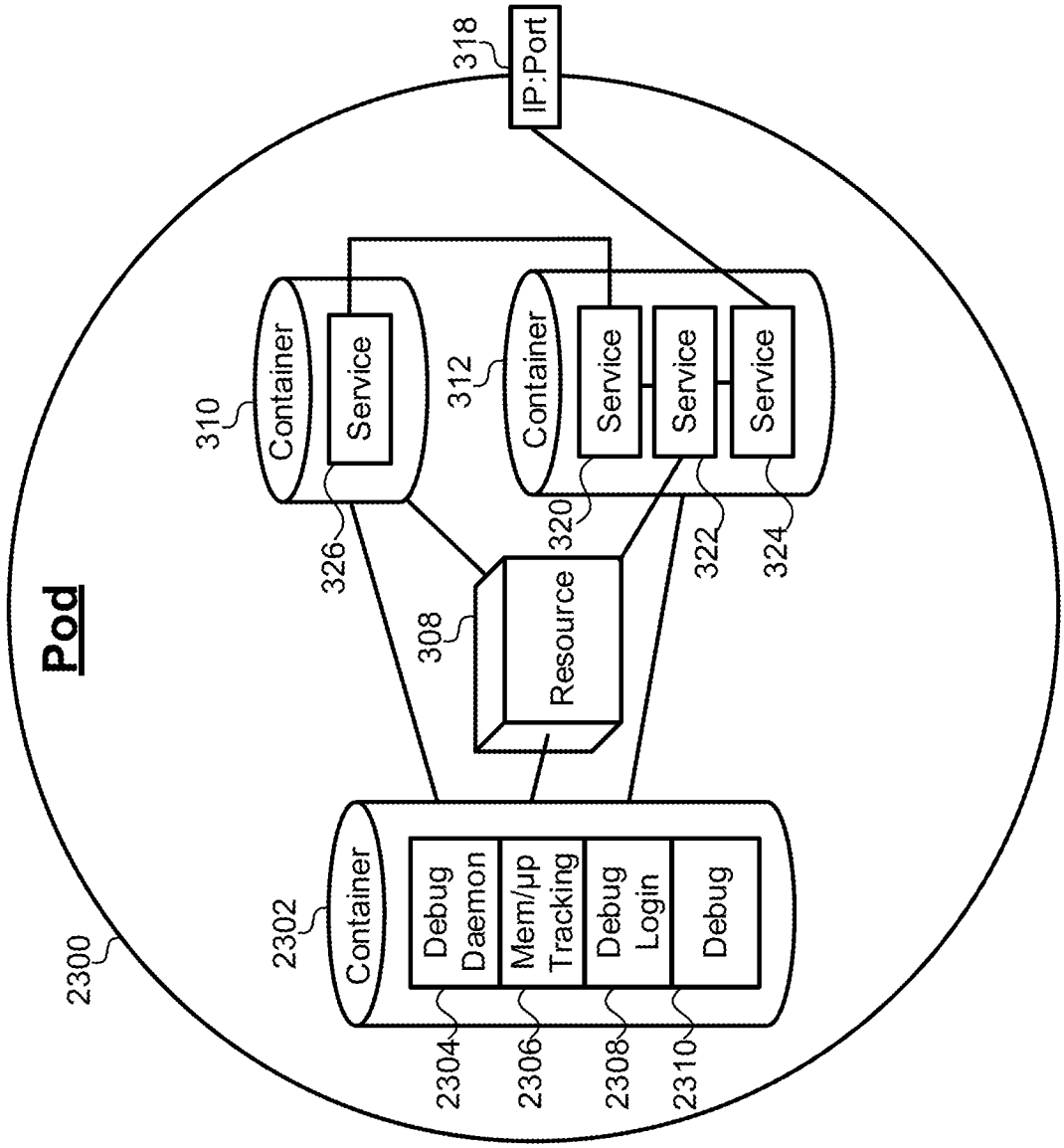
Pod
2300
318
IP:Port
310
Container
Service
312
Container
Service
Service
Service
326
320
322
324
308
Resource
2302
Container
Debug Daemon
Mem/µp Tracking
Debug Login
Debug
2304
2306
2308
2310

FIG. 23

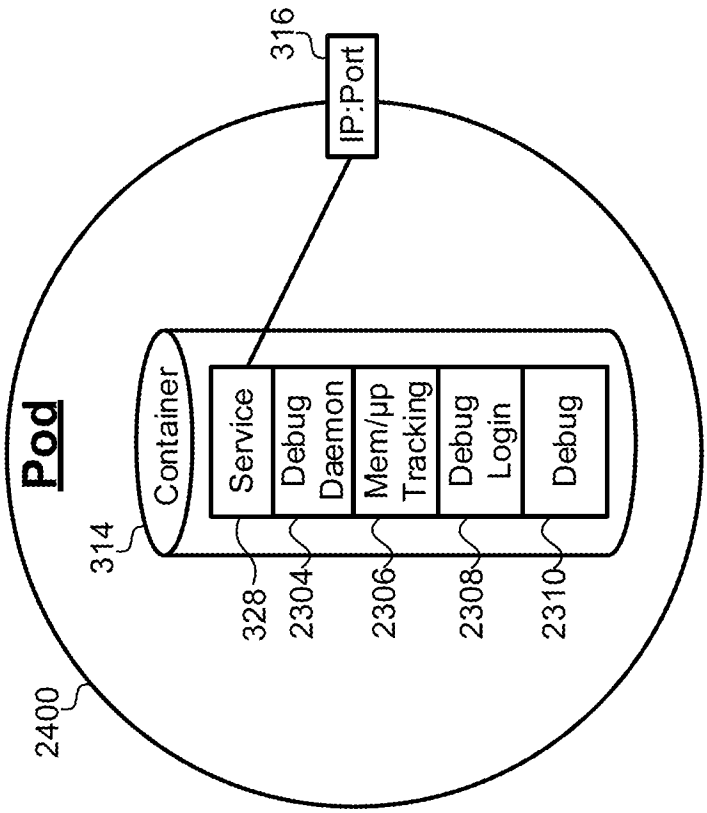
316
IP:Port
Pod
Container
Service
Debug Daemon
Mem/µp Tracking
Debug Login
Debug
314
328
2304
2306
2308
2310
2400

FIG. 24

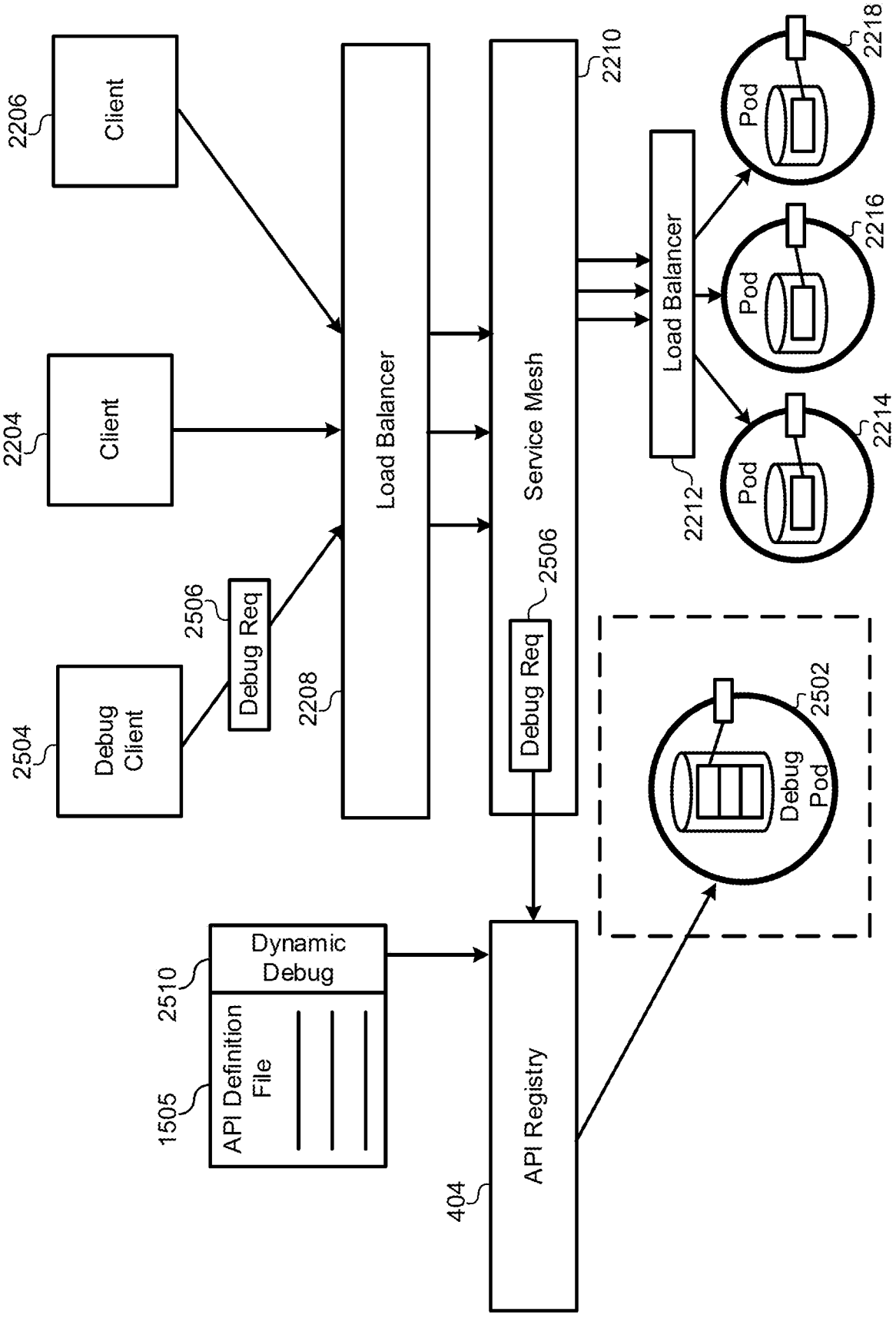
Client
2206
Client
2204
Debug Client
2504
Debug Req
2506
2208
Load Balancer
Service Mesh
2210
Debug Req
2506
Load Balancer
2212
Pod
2218
Pod
2216
Pod
2214
Debug Pod
2502
Dynamic Debug
2510
API Definition File
1505
API Registry
404

FIG. 25

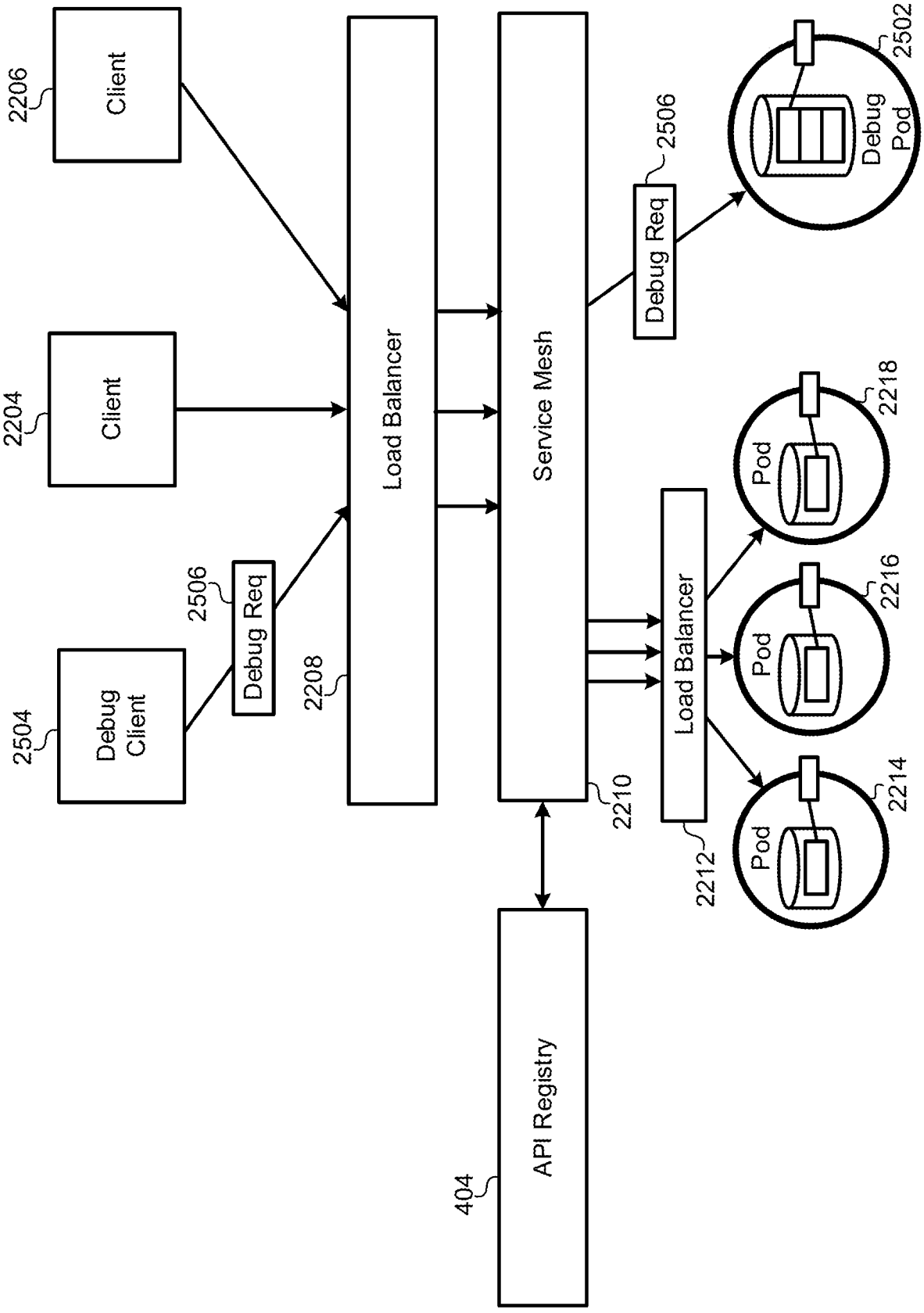
Client
2206
Client
2204
Debug Client
2504
Debug Req
2506
2208
Load Balancer
Service Mesh
2210
Debug Req
2506
Debug Pod
2502
Load Balancer
2212
Pod
2218
Pod
2216
Pod
2214
API Registry
404

FIG. 26

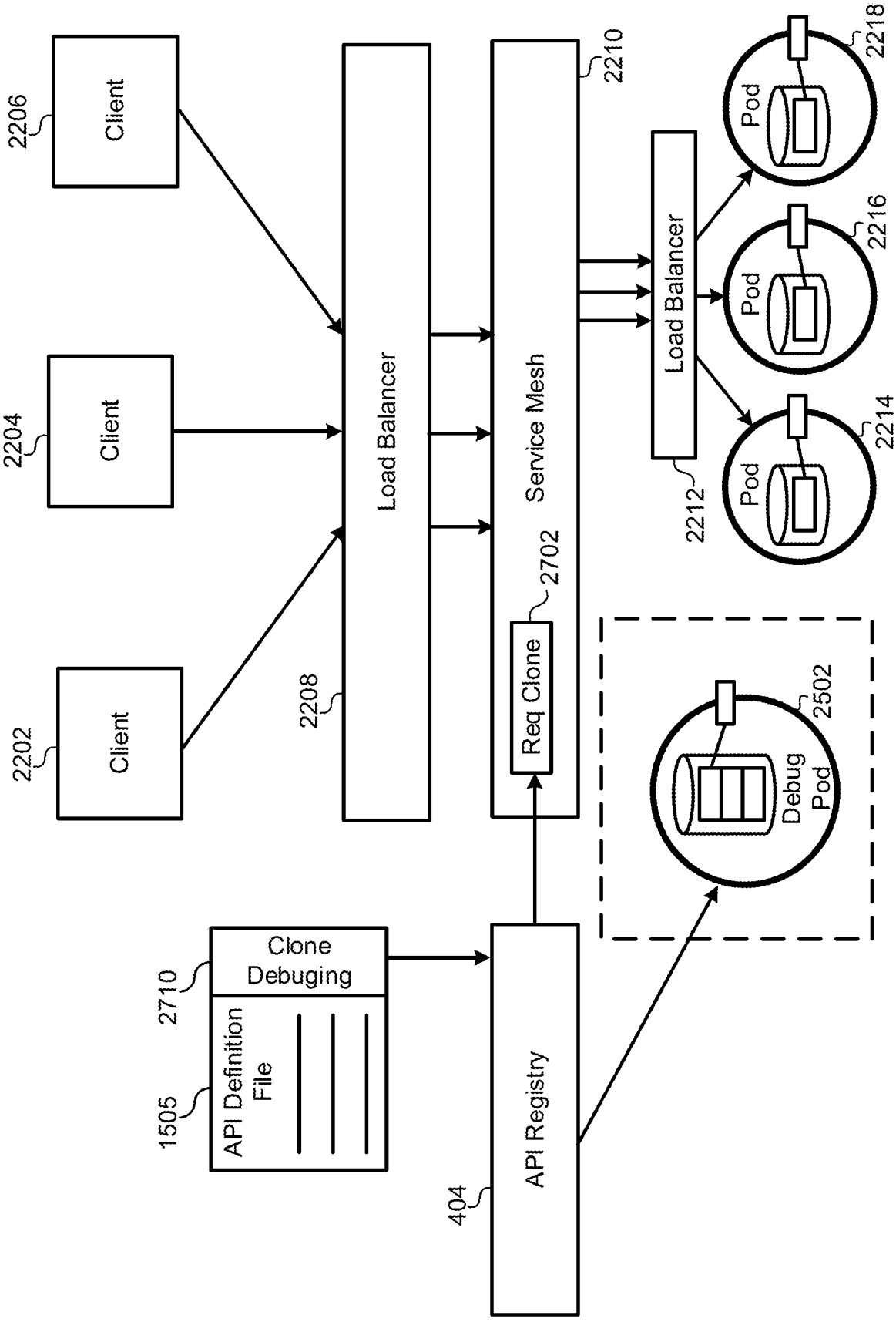
2206
Client
2204
Client
2202
Client
2208
Load Balancer
2210
Service Mesh
2702
Req Clone
2212
Load Balancer
2218
Pod
2216
Pod
2214
Pod
2502
Debug Pod
2710
Clone Debuging
1505
API Definition File
404
API Registry

FIG. 27

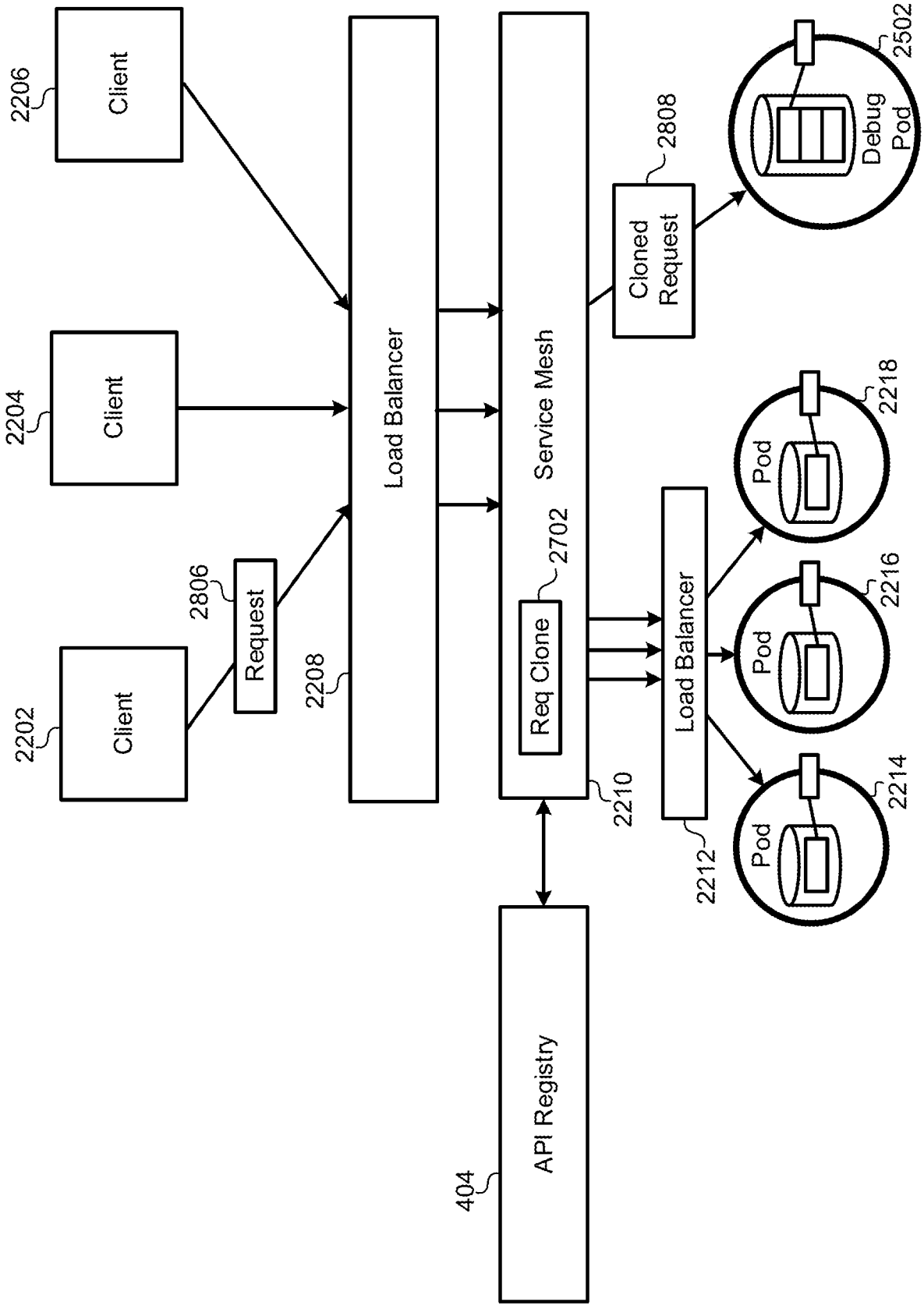
2206
Client
2204
Client
2202
Client
2806
Request
2208
Load Balancer
Service Mesh
2702
Req Clone
2210
API Registry
404
2808
Cloned Request
2502
Debug Pod
2218
Pod
2216
Pod
2214
Pod
Load Balancer
2212

FIG. 28

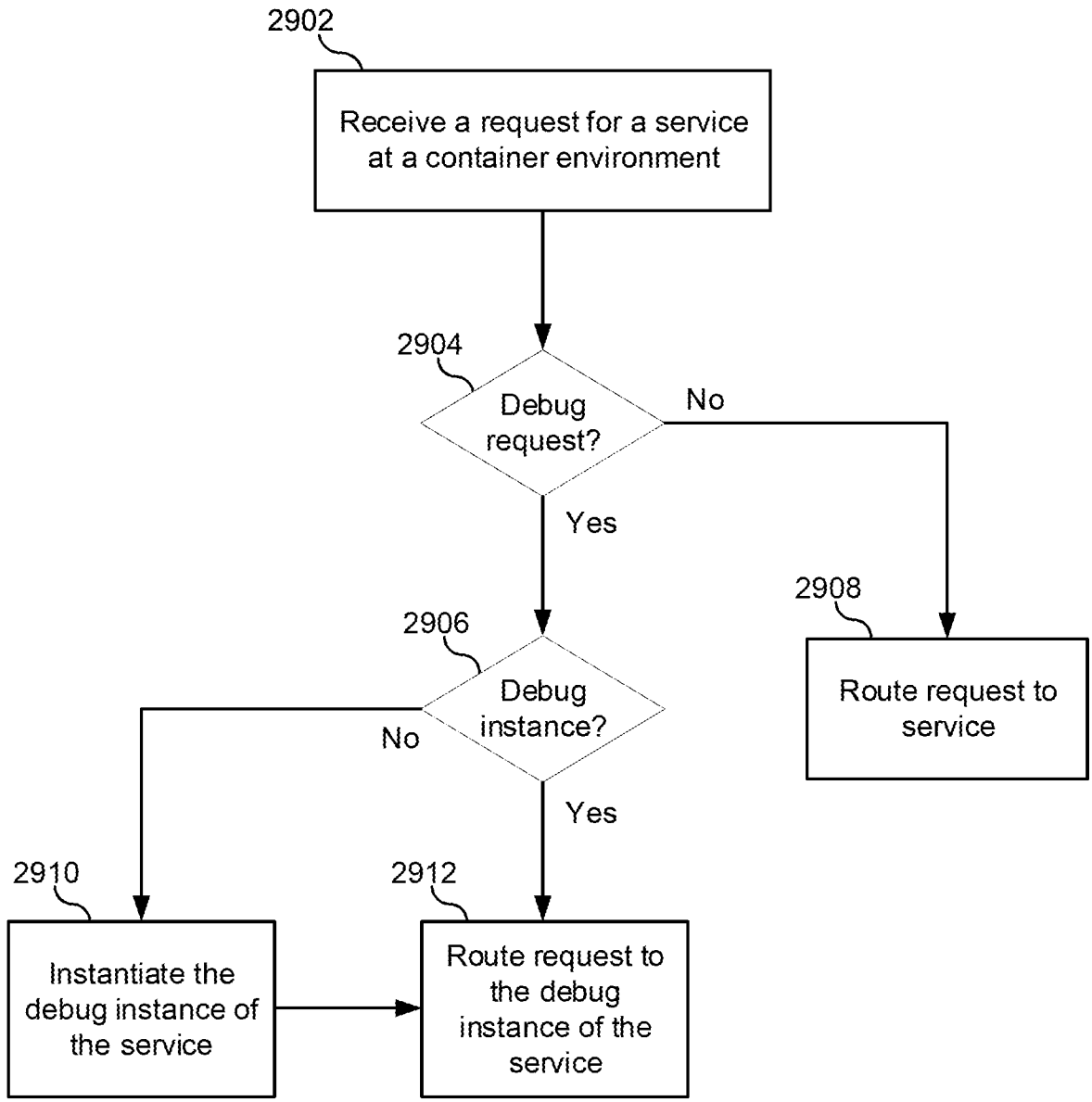
2902
Receive a request for a service at a container environment
2904
Debug request?
No
Yes
2908
Route request to service
2906
Debug instance?
No
Yes
2910
Instantiate the debug instance of the service
2912
Route request to the debug instance of the service

FIG. 29

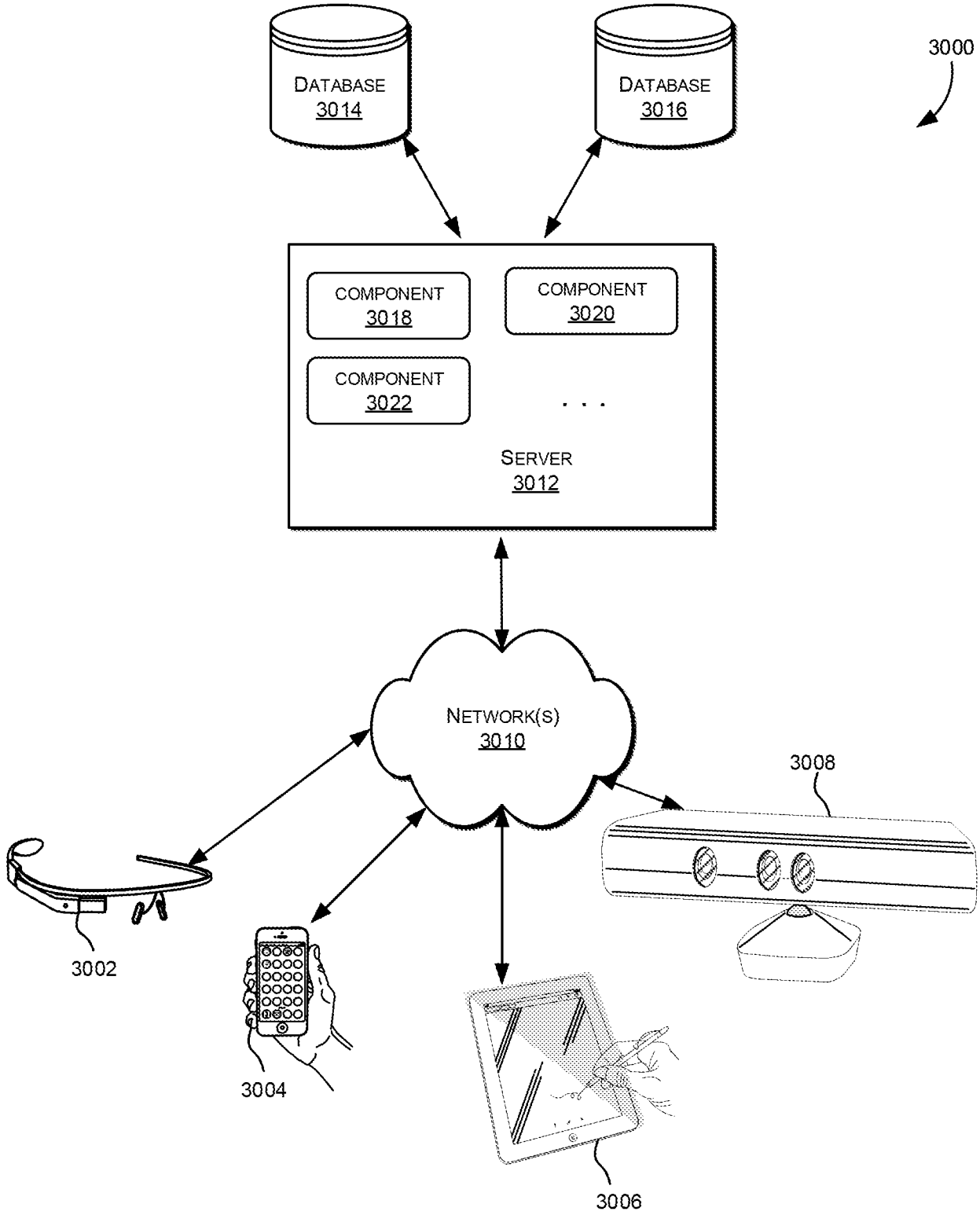
3000
DATABASE
3014
DATABASE
3016
COMPONENT
3018
COMPONENT
3020
COMPONENT
3022
. . .
SERVER
3012
NETWORK(S)
3010
3008
3002
3004
3006

FIG. 30

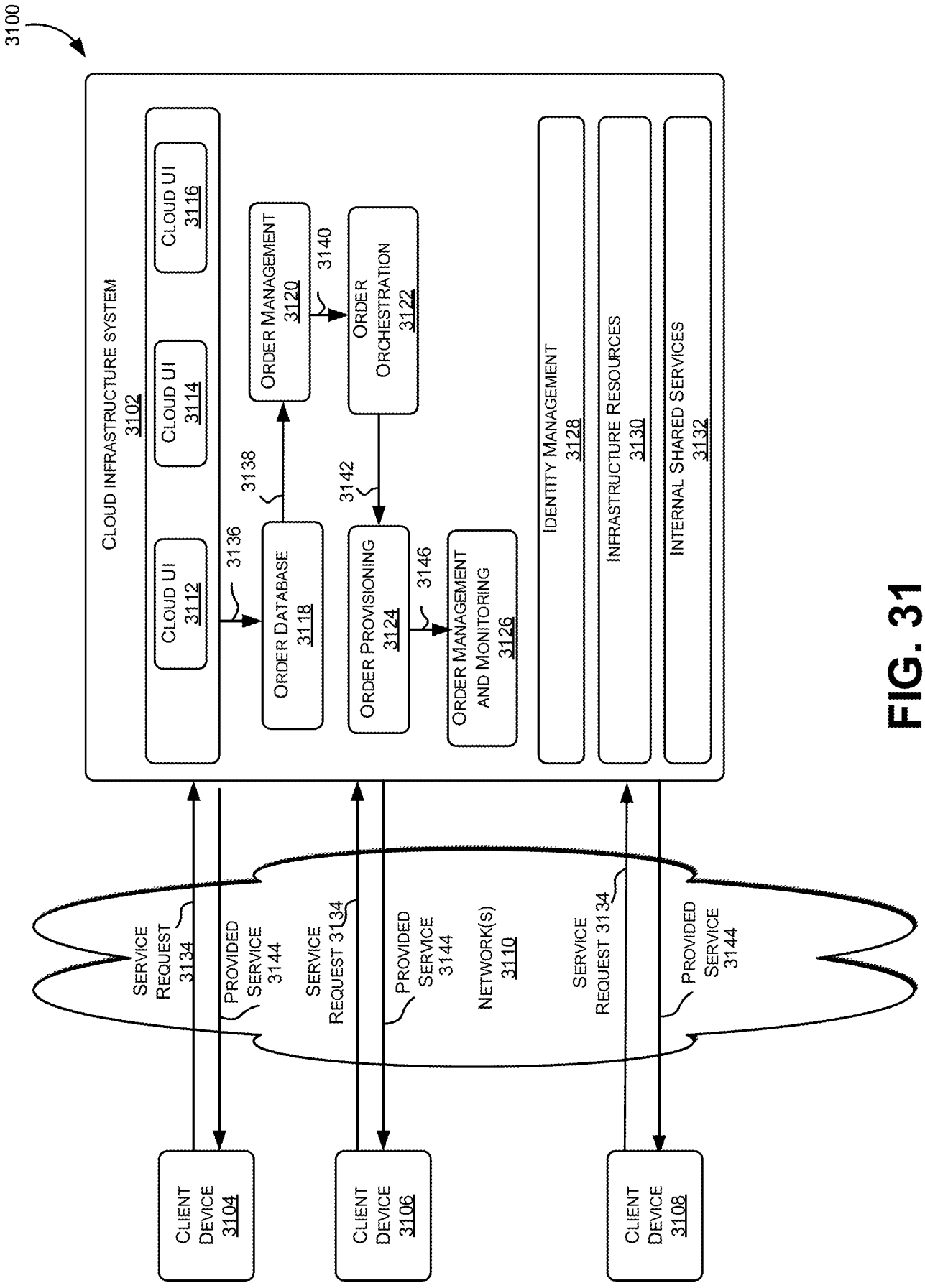
3100
CLOUD INFRASTRUCTURE SYSTEM
3102
CLOUD UI
3112
CLOUD UI
3114
CLOUD UI
3116
ORDER DATABASE
3118
ORDER MANAGEMENT
3120
ORDER
ORCHESTRATION
3122
ORDER PROVISIONING
3124
ORDER MANAGEMENT
AND MONITORING
3126
IDENTITY MANAGEMENT
3128
INFRASTRUCTURE RESOURCES
3130
INTERNAL SHARED SERVICES
3132
3136
3138
3140
3142
3146
SERVICE
REQUEST
3134
PROVIDED
SERVICE
3144
NETWORK(S)
3110
CLIENT
DEVICE
3104
CLIENT
DEVICE
3106
CLIENT
DEVICE
3108

FIG. 31

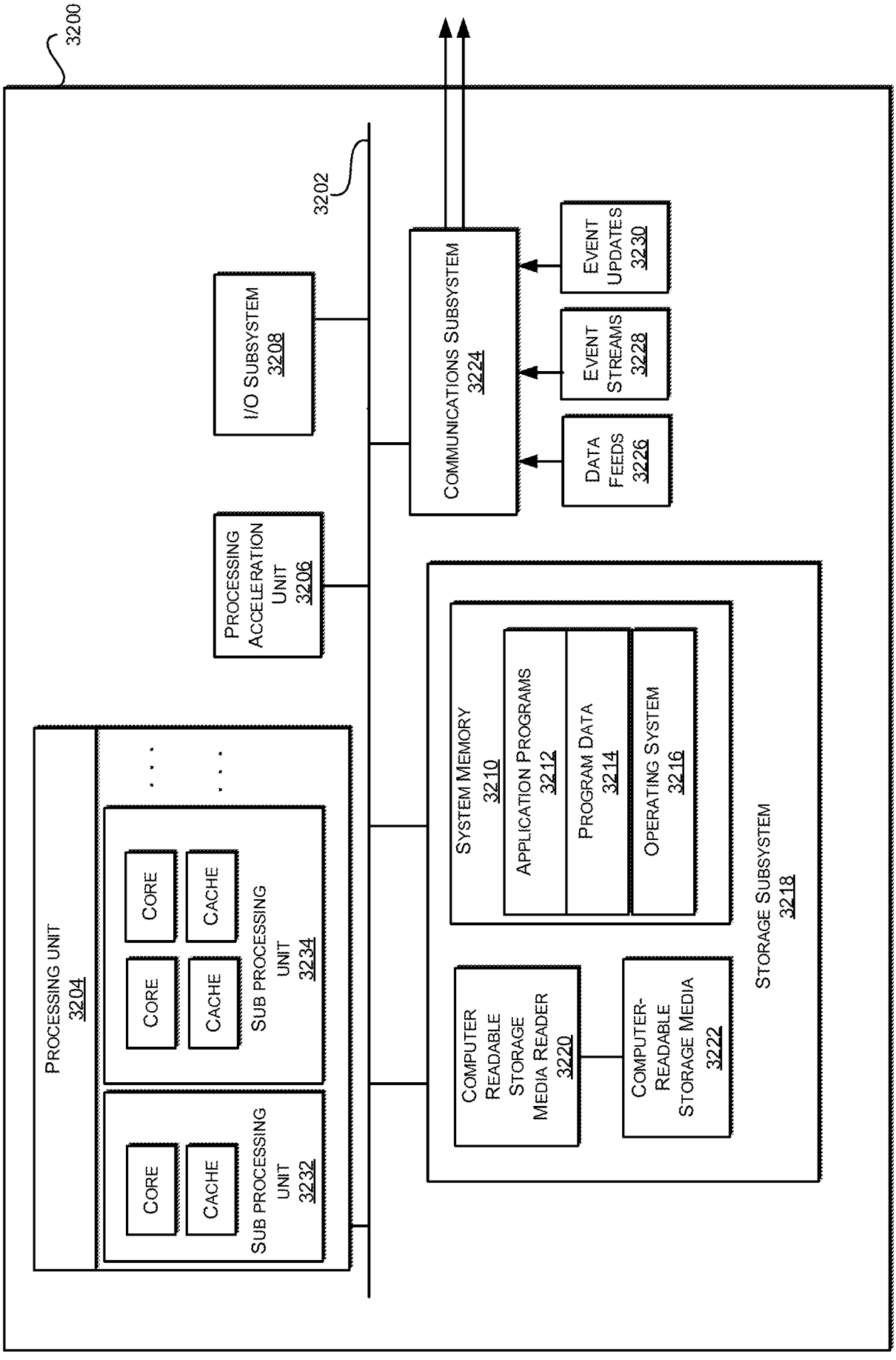
3200
3202
I/O SUBSYSTEM 3208
COMMUNICATIONS SUBSYSTEM 3224
EVENT UPDATES 3230
EVENT STREAMS 3228
DATA FEEDS 3226
PROCESSING ACCELERATION UNIT 3206
PROCESSING UNIT 3204
CORE
CACHE
SUB PROCESSING UNIT 3234
SUB PROCESSING UNIT 3232
SYSTEM MEMORY 3210
APPLICATION PROGRAMS 3212
PROGRAM DATA 3214
OPERATING SYSTEM 3216
COMPUTER READABLE STORAGE MEDIA READER 3220
COMPUTER-READABLE STORAGE MEDIA 3222
STORAGE SUBSYSTEM 3218

FIG. 32

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2018/053628

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F11/36 G06F8/60 G06F9/50
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2017/046146 A1 (JAMJOOM HANI T [US] ET AL) 16 February 2017 (2017-02-16) abstract paragraph [0062] - paragraph [0081] -----	1-20
A	LIU DESHENG ET AL: "CIDE: An Integrated Development Environment for Microservices", 2016 IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING (SCC), IEEE, 27 June 2016 (2016-06-27), pages 808-812, XP032953859, DOI: 10.1109/SCC.2016.112 [retrieved on 2016-08-31] the whole document -----	1-20

-/--

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
3 January 2019	10/01/2019

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Renault, Sophie

2

Form PCT/ISA/210 (second sheet) (April 2005)

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	KANG HUI ET AL: "Container and Microservice Driven Design for Cloud Infrastructure DevOps", 2016 IEEE INTERNATIONAL CONFERENCE ON CLOUD ENGINEERING (IC2E), IEEE, 4 April 2016 (2016-04-04), pages 202-211, XP032908141, DOI: 10.1109/IC2E.2016.26 [retrieved on 2016-06-01] the whole document -----	1-20

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2018/053628

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2017046146 A1	16-02-2017	NONE	