April 2023

DYNAMIC RESOURCE PROFILE-BASED CONTAINER AND SERVERLESS CONSTRUCT FOR COMPOSABLE OBSERVABILITY TRACING

Nagendra Kumar Nainar

Akram Sheriff

Rajesh I V

Carlos Pignataro

Dave Zacks

DYNAMIC RESOURCE PROFILE-BASED CONTAINER AND SERVERLESS CONSTRUCT FOR COMPOSABLE OBSERVABILITY TRACING

AUTHORS:
Nagendra Kumar Nainar
Akram Sheriff
Rajesh I V
Carlos Pignataro
Dave Zacks

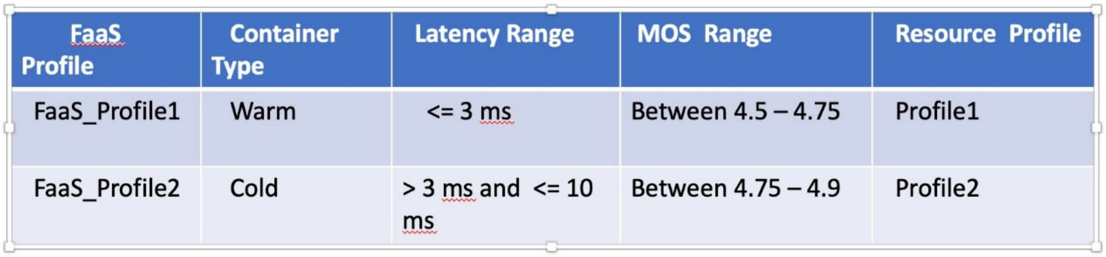
ABSTRACT

As a result of recent industry attention towards full-stack observability, there are various application performance monitoring (AMP) tools, runtime application self-protection (RASP) tools, and distributed tracing modules (such as OpenTelemetry (OTel), the Distributed Application Runtime (Dapr), etc.) available for different programming languages which may be injected as instrumentation code into an actual application to support a very detailed trace or log collection for subsequent analysis. Using a no- or low-code construct, the observability instrumentation code and the application code may be dynamically bundled together to create the function that is required to execute the relevant transactions. Techniques are presented herein that support a new dynamic, traffic-aware Composable Function that leverages an inbound traffic request (comprising metadata and various attributes with, for example, the relevant tracing context header, custom Hypertext Transfer Protocol Secure (HTTPS) header, or OTel header with baggage) to identify and determine whether a function bundle should be composed with observability, APM, or RASP tools for the efficient utilization of the resource. Such a dynamically composed function may, in turn, be hosted as one or more containers within a pod or as a serverless function to provide more flexibility across different application and cluster environments.

DETAILED DESCRIPTION

In order to realize the maximum benefits that may arise from virtualization and a cloud offering, many applications are adopting the use of a cloud-native and/or a serverless construct for functional executions. Any application may be split into microservices for individual development and then instantiated as containers or functions that leverage Kubernetes or a serverless construct to host and execute the function.

When using a Kubernetes architecture, such microservices may be hosted as multiple containers within one or more pods and may use the relevant container networking and/or ingress rules for load balancing and distribution (as depicted in Figure 1, below).



FaaS Profile	Container Type	Latency Range	MOS Range	Resource Profile
FaaS_Profile1	Warm	<= 3 ms	Between 4.5 – 4.75	Profile1
FaaS_Profile2	Cold	> 3 ms and <= 10 ms	Between 4.75 – 4.9	Profile2

Figure 1: Exemplary Arrangement

A serverless construct leverages a running function as a service (FaaS) to execute the function without any dedicated computing resources. Instead, the functions are only instantiated when there is an incoming trigger (such as a Hypertext Transfer Protocol (HTTP) request, a Transmission Control Protocol (TCP) synchronize (SYN) message, an application programming interface (API) request, etc.) which, in turn, triggers the packaging together of the function's code and its dependencies (e.g., binaries and libraries) for an on-the-fly dynamic resource allocation and execution. Figure 2, below, depicts elements of such a process.

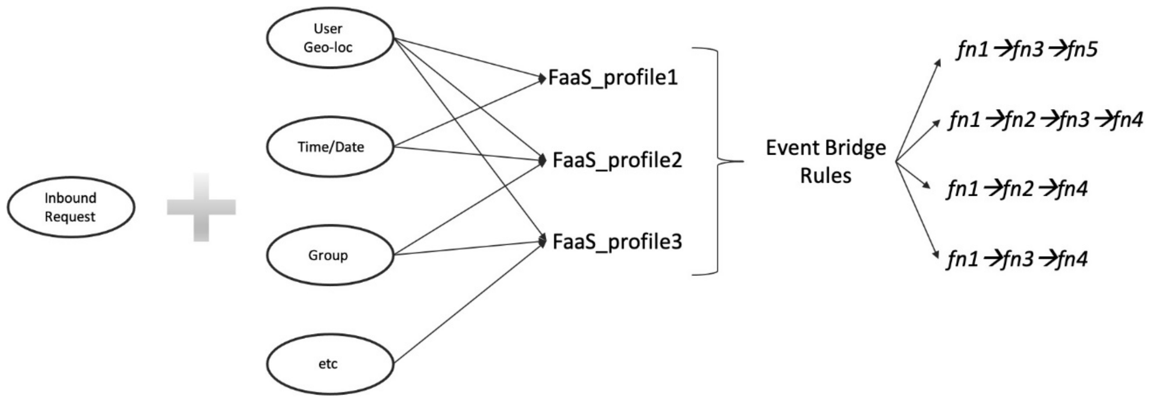


Figure 2: Illustrative Process

With the recent attention of the industry towards full-stack observability, there are various application performance monitoring (AMP) tools, runtime application self-protection (RASP) tools, and distributed tracing modules (such as OpenTelemetry (OTel), the Distributed Application Runtime (Dapr), etc.) available for different programming languages which may be injected as instrumentation code into an actual application for a very detailed trace or log collection for subsequent analysis. Using a no- or low-code construct, the observability instrumentation code and the application code may be dynamically bundled together to create the function that is required to execute the relevant transactions.

Figure 3, below, illustrates elements of the process that was described above.

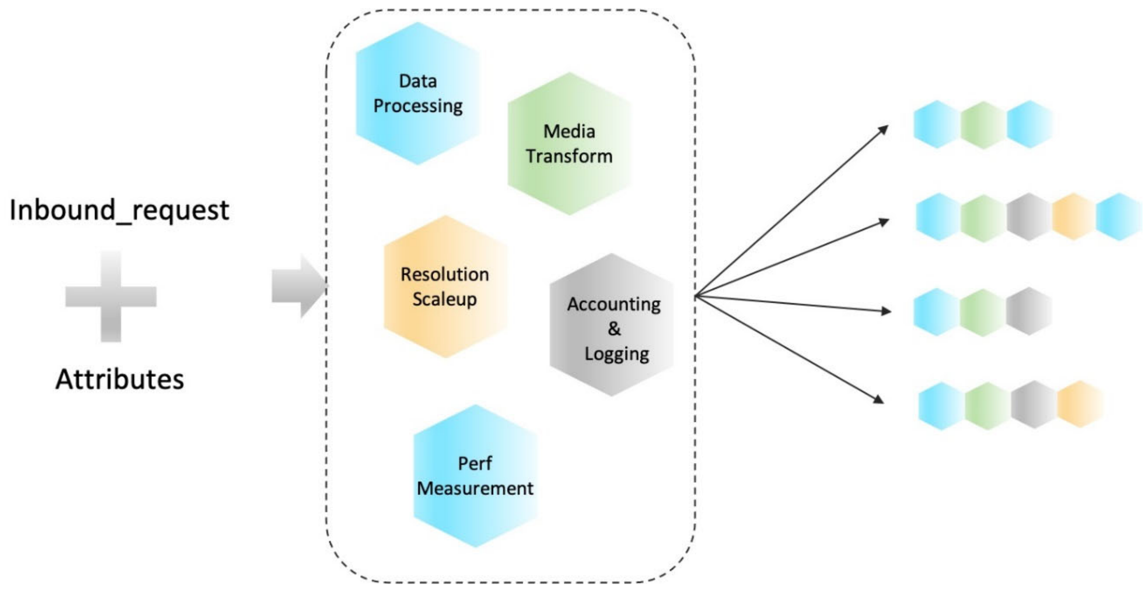


Figure 3: Exemplary Process

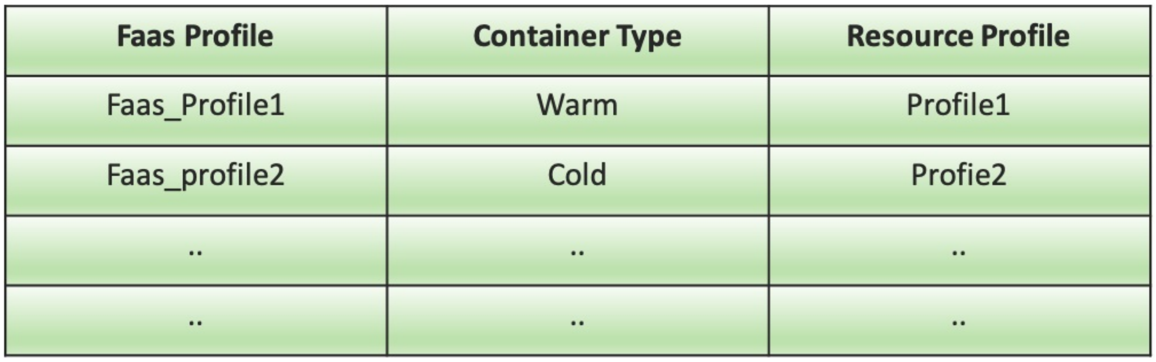
As shown in Figure 3, above, an application may also exist with hybrid agents such as Dapr or OTel to tailor the customer-specific use cases. Running all of the modules (as part of a container) or bringing all of the lambda functions together in a cloud may not be a significant issue from a resource allocation point of view. However, such an approach obviously involves additional cost. But while applying the same approach in other resource constrained environments (such as edge computing, cloud edge, or an Internet of things (IoT) environment), applying the right resource requirement is key from both an efficiency and cost point of view.

To address the types of challenges that were described above, techniques are presented herein that support a new dynamic, context-aware Composable Function that leverages an inbound traffic request (with the relevant tracing context header) to identify whether a function bundle should be composed with an observability, APM, or RASP tool for the efficient utilization of the resource. Such a composed function may, in turn, be hosted as containers within a pod or as a serverless function.

As described above, a Composable Function according to the techniques presented herein leverages an inbound traffic request (comprising metadata and attributes with the relevant tracing context header, custom HTTP Secure (HTTPS) header, or OTel header with baggage) to identify and determine whether a function bundle should be composed with an observability, APM, or RASP tool for the efficient utilization of the resource. Such a dynamically-composed function may, in turn, be hosted as one or more containers within a pod or as a serverless function to provide more flexibility across different application and cluster environments.

For clarity and simplicity of exposition, the following narrative, which presents a detailed description of the techniques presented herein, is split into two parts. The first part covers the presented techniques operating within a Kubernetes® environment while the second part covers the presented techniques operating within a serverless construct. Kubernetes® and K8S® are registered trademarks of The Linux Foundation.

Within a Kubernetes environment, the control plane component may be configured (e.g., using a yet another markup language (YAML) file or some other dynamic approach) to host the application alone as a container of resource profile X or by bundling it together with additional APM or RASP components as a container with resource profile Y. Figure 4, below, presents elements of various illustrative profiles.



Faas Profile	Container Type	Resource Profile
Faas_Profile1	Warm	Profile1
Faas_profile2	Cold	Profie2
..	..	..
..	..	..

Figure 4: Illustrative Profiles

As shown in Figure 4, above, different combinations of modules may be associated with different resource profiles which are used while hosting the containers (as a pod or within a pod). Under one approach, the autoscaler component of Kubernetes may be augmented to dynamically scale up or down using different profile-based containers to handle different types of traffic or for distributed tracing. The above approach is applicable when the APM or RASP modules are integrated as agents in the application or host. Alternately, it is also possible to instantiate the APM or RASP modules as a sidecar that is injected alongside the application container. In such a scenario, the table that was shown in Figure 4, above, may indicate for a profile the different sidecars (i.e., modules). In either of the two cases, a profile will identify the type of resources that are required to host an application and the modules.

It normally takes at most a few milliseconds (msec) to host a container and the techniques presented herein leverage that capability. Different types of triggers may be used to initiate this dynamic multi-profile container scaling. Such triggers may include, for example, a time-based trigger, a source user or site-based trigger, an application header or application identifier (AppID)-based trigger, or an Internet Protocol (IP) service-level agreement (SLA) attribute-based Trigger.

For a time-based trigger, the local time may be used as a means to launch a container with an application and the relevant APM modules. For example, a customer may notice that during a specific time an application is not performing well and, as a result, would like to obtain more details (such as performance, load, etc.) for analysis. The specific time may be used as a trigger where one or two instances are hosted with the application along with the APM or RASP modules for performance measurement. To handle the application load and to perform the measurements, the customer may host two instances of the application (with virtual central processing unit (vCPU)=2 and Mem=1 gigabyte (G)) and one instance of the application along with an APM of type Dapr and an APM of type OTel (with vCPU=6 and Mem=2G) instead of three instances with six vCPU and 2G of memory. This allows for scalability and the efficient use of the available resources.

For a source user or site-based trigger, the inbound traffic source information may be used to identify if an instance with a different profile is required. For example, users who are accessing an application from a corporate network may not need biometric or real-time authentication, while the remote users who are accessing the application from an external network may require such services. Based on the user information, the relevant container with a RASP module may be hosted along with the application-only instances.

For an application header-based trigger, the presence of a trace context in the application header may be used as a trigger. It is important to note that the initial request or GET probe may carry the trace context that may need to trigger the span for tracing. Simple caching may be used to launch the container and forward such a request.

For a serverless environment, the techniques presented herein augment the request to a function table to be more granular by considering additional attributes and map the same to the relevant modules that need to be combined together. Figure 5, below, presents elements of an example that illustrates such an approach.

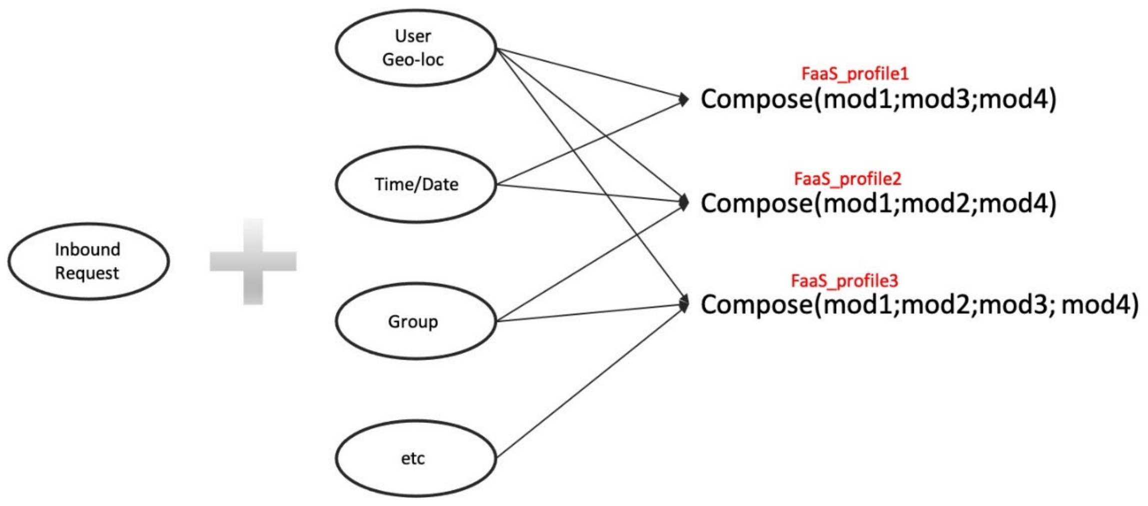
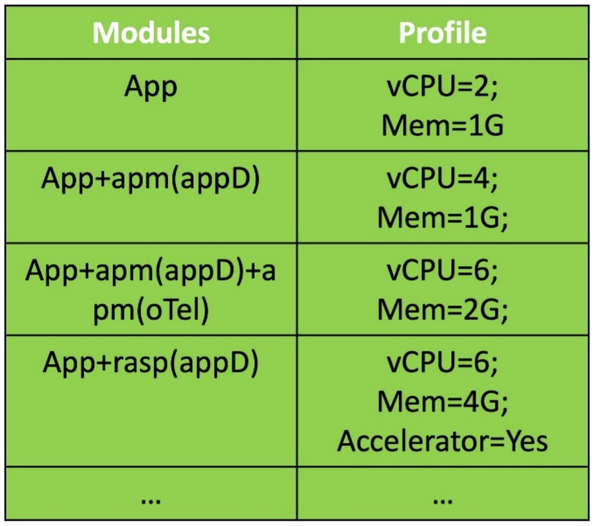


Figure 5: Exemplary Combinations

Depending upon the combination of the inbound request type and the additional attributes, the selection of a module may be influenced. As shown in the example in Figure 5, above, if an inbound request includes additional cookies that identify peak-hour time and originating users from a specific geolocation, then mod1, mod3, and mod4 may be combined together to create the function code. The composed function may include mod1 which is the actual application in addition to different hybrid agents mod3 (e.g., Dapr) and mod4 (e.g., OTel).

For each such combination of modules, the table may be further augmented to encompass different types of computing resources, as depicted in Figure 6, below.



Modules	Profile
App	vCPU=2; Mem=1G
App+apm(appD)	vCPU=4; Mem=1G;
App+apm(appD)+apm(oTel)	vCPU=6; Mem=2G;
App+rasp(appD)	vCPU=6; Mem=4G; Accelerator=Yes
…	…

Figure 6: Exemplary Augmented Table

According to the techniques presented herein, depending upon the type of inbound request, an FaaS profile may be identified dynamically and the relevant resource profile may be used to identify the container or compute resource that is to be used to execute the composed function code. For example, the lambda or function router may leverage multiple attributes that are associated to an inbound request (such as a user profile, a location, a time and date, etc.) to identify if the application must be simply injected into a warm or cold container or if the application must be composed with a RASP module for execution.

According to further aspects of the techniques presented herein, the inbound synthetic probe or packet may be used as a trigger to identify that the FaaS profile that is to be used is a cold container and has a low resource profile. It may be noted that such synthetic probes cannot be used for performance measurement. Instead, they can be used for other metrics, log, or event collection (such as functional validation).

It is important to note that while the above narrative was primarily positioned and explained with the use of multiple agents, the techniques presented herein may also be generalized to include additional components. Connecting the dots together with the example that was presented above, elements of a Composable FaaS (cFaaS) are shown in Figure 7, below.

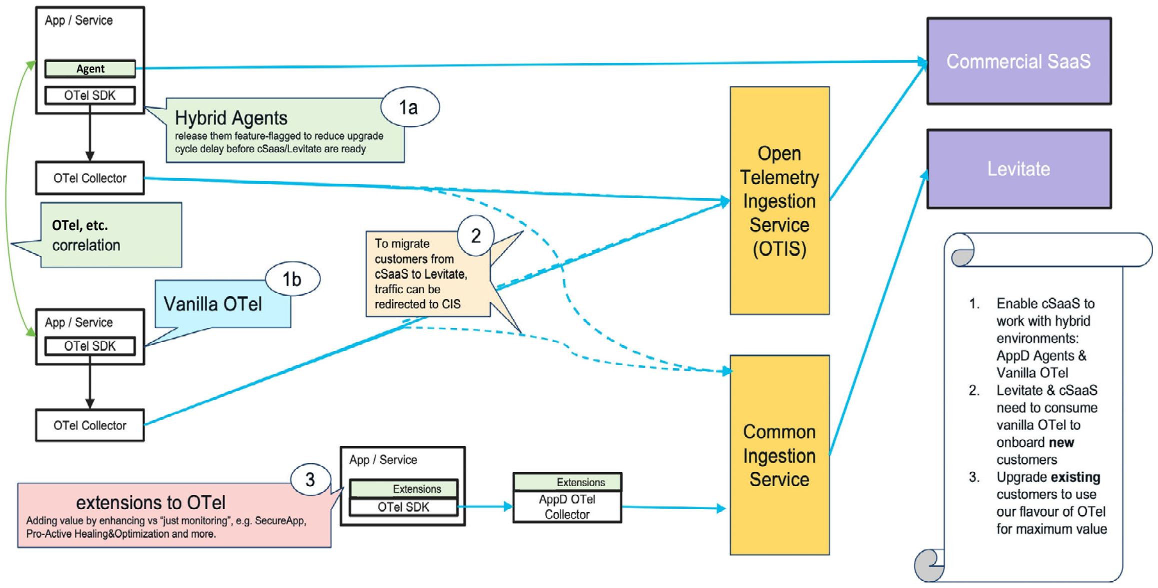

App / Service
Agent
OTel SDK
OTel Collector
Hybrid Agents
release them feature-flagged to reduce upgrade cycle delay before cSaas/Levitate are ready
1a
OTel, etc. correlation
Vanilla OTel
1b
App / Service
OTel SDK
OTel Collector
To migrate customers from cSaaS to Levitate, traffic can be redirected to CIS
2
extensions to OTel
Adding value by enhancing vs "just monitoring", e.g. SecureApp, Pro-Active Healing&Optimization and more.
3
App / Service
Extensions
OTel SDK
Extensions
AppD OTel Collector
Open Telemetry Ingestion Service (OTIS)
Common Ingestion Service
Commercial SaaS
Levitate
1. Enable cSaaS to work with hybrid environments: AppD Agents & Vanilla OTel
2. Levitate & cSaaS need to consume vanilla OTel to onboard new customers
3. Upgrade existing customers to use our flavour of OTel for maximum value

Figure 7: Illustrative cFaaS

While the above approach illustrates how such a concept may be used to dynamically create a Composable Function's code and invoke the same using the relevant required resources, the concept may also be extended for a daisy-chained FaaS using an event bridge.

Under such an arrangement, each module may be considered as a standalone function code and the event bridge rules may be used to dynamically identify the modules that should be invoked and the order in which the modules should be daisy-chained together. Figure 8, below, presents elements of such an approach.

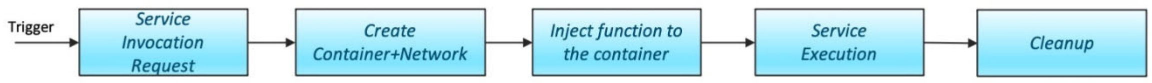


Figure 8: Exemplary Daisy Chaining

Each FaaS profile (i.e., module combination) may be treated as different function codes that are executed in a specific order. As described above, the FaaS profiles may be mapped to event bridge rules that define which FaaS function should be executed from a daisy-chained FaaS function and in what sequence order.

Aspects of the techniques presented herein also support the use of IP SLA attributes which may be monitored by a cFaaS tool entity to trigger the dynamic creation and selection of different FaaS templates at runtime based on the IP SLA parameters. For example, different cFaaS profiles may be instantiated for different kinds of traffic patterns such as an Internet Control Message Protocol (ICMP) Echo operation (for wide area network (WAN) round-trip time (RTT) monitoring), an ICMP Path Echo operation (for a traceroute or hop-by-hop monitoring), a User Datagram Protocol (UDP) jitter operation (for Voice over Internet Protocol (VoIP) monitoring), or a video operation (for video monitors).

Figure 9, below, presents elements of the type of approach that was described above.

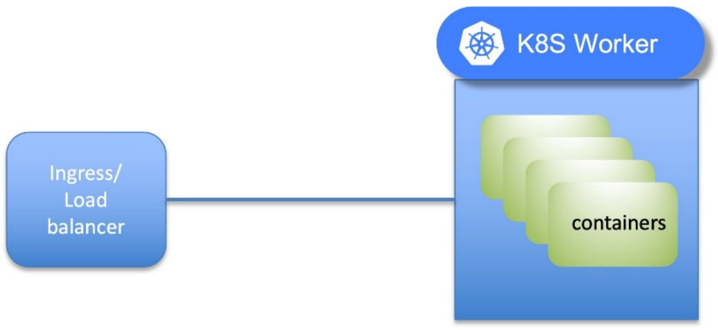


Figure 9: Exemplary Arrangement

In summary, techniques have been presented herein that support a new dynamic, traffic-aware Composable Function that leverages an inbound traffic request (comprising metadata and various attributes with, for example, the relevant tracing context header, custom HTTPS header, or OTel header with baggage) to identify and determine whether a function bundle should be composed with observability, APM, or RASP tools for the efficient utilization of the resource. Such a dynamically composed function may, in turn, be hosted as one or more containers within a pod or as a serverless function to provide more flexibility across different application and cluster environments.