針對 Microservices 應用程式之以代理人為基底的監控機制
Agent-based Monitoring Mechanism for Microservices Applications

陳錫民 薛念林 陳奕中 李霽烝 陳仰萱
His-Min Chen, Nien ien-Lin Hsueh, Yi-Chung Chen, Lee Chi-Chen, Yang-Syuan Chen
逢甲大學 資訊工程學系

Email: {hsiminc, nlhsueh, chenyic}@fcu.edu.tw,
rree8484@yahoo.com.tw, yangsyuan8012@gmail.com

摘 要

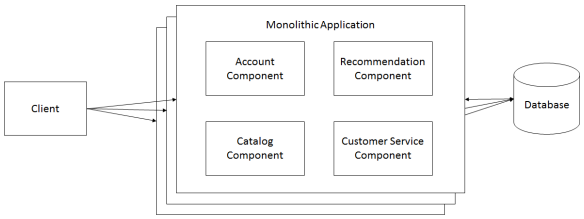
相較於傳統 Monolithic 軟體架構，Microservices 軟體架構具有縮短產品開發時程、適應軟體需求快速改變、增加軟體開發產能與提升故障隔離性等優點。然而 Microservices 架構中構成應用程式的軟體元件是以微型服務的形式部署在分散的雲端環境中，雖然具有彈性，另一方面也意味著微型服務的監控與管理更為複雜與困難。對微型服務應用程式之供應者而言，如何了解微型服務應用程式整體之營運狀況與快速地找出營運問題所在已成為重要的議題。因此，在本研究中，我們提出了一個利用軟體代理人與 ElasticSearch、Logstash 和 Kibana (ELK) Stack 之整合的監測機制，幫助應用程式供應者了解他們部署在雲端上由微型服務所構成的應用程式之營運狀態。此機制採用軟體代理人與 Logstash 收集、過濾和轉換來自各種微型服務的監測度量資料，並將處理後的資料存儲於 ElasticSearch 分散式的資料庫中。此外，我們提出了一個 Kibana 儀表板方便使用者可以輕鬆地透過視覺化分析圖表了解應用程式之營運狀況。

關鍵字： Microservices Architecture、Quality of Service、Service Monitoring、Software Agent、ELK Stack、Docker Container

一、緒論

近年來由於各式各樣電子裝置的普及，軟體應用程式的發展除了面臨到使用者需求的快速改變外，為了吸引使用者長期附著在軟體上，軟體功能必須不斷地推陳出新。為了因應這樣的改變，我們可以發現越來越多的軟體廠商在軟體開發流程上紛紛採用能快速適應需求改變與更新頻率的敏捷方法[1,2]以取代原本的傳統瀑布式開發方法。

然而在軟體架構(Software Architecture)方面，卻沒有隨著這樣的改變有突破性解決方法提出，過去企業級軟體設計大都採用三層式的軟體架構[3]，系統主要由(1)客戶端程式、(2)伺服器端商業邏輯(Business Logic)與(3)底層資料庫所構成。伺服器端商業邏輯負責處理客戶端請求、執行商業邏輯、連結資料庫存取資料、最後將結果回覆給客戶端程式。若將整體伺服器端商業邏輯包裝成一個單一的可執行單元，這樣的架構稱為 Monolith 應用程式[4,5]。如圖一所示，這是一個常見的電子商務網站架構規劃，商業邏輯層中所有的軟體元件是由相同的程式語言所開發並且部署在伺服器由單一執行程序(Process)所執行。隨著客戶端請求的增加，可以利用雲端技術，對商業邏輯層應用程式生成多份複本(Replica)，同時結合平衡負載機制[6]做水平性的擴展(Horizontal Scaling)，以服務大量的客戶端請求。



圖一 基於 Monolithic 架構的電子商務網站規劃

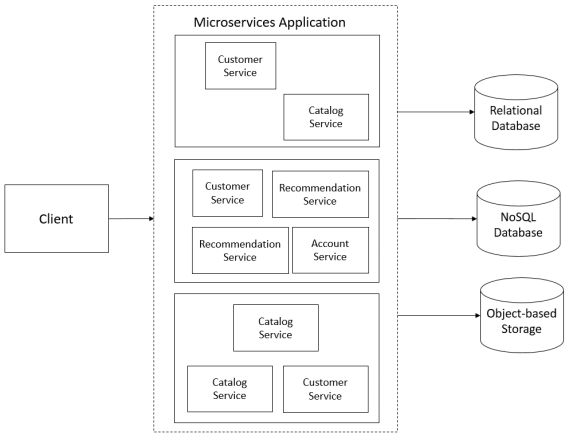
Monolithic 架構的優點是(1)對程式開發人員而言，整個中間層應用程式只有一個程式基底(codebase)容易開發與部署、(2)容易做水平性的擴展與(3)對營運人員而言，系統容易維護與管理。然而隨著企業組織與軟體的成長，採用 Monolithic 架構容易造成下列問題：(1)每一個軟體元件沒有明確的負責人員、(2)軟體元件之間的耦合性會越來變得越緊密、(3)不能根據需求特性採用合適的程式語言、(4)水平性擴展是無差別的擴展，不能針對特定的軟體元件擴展以及(5)當其中一個軟體元件發生問題時整體軟體系統也會隨著無法運作。另一方面為了因應客戶需求的改變或是程式缺陷修復，只要是任何一個軟體元件做微小的更改，整體應用程式必須被重新建構、測試與部署，拉長了軟體開發時程。

為此，在 2014 年 James Lewis 與 Martin Fowler 提出了 Microservices 軟體架構[4,5]觀念。Microservices 是種架構模式(architectural pattern)，其目的是將一軟體應用程式分解成多個可獨立開發、測試、執行與部署的微型服務，微型服務可以根據 Single Responsibility Principle[1]方法界定服務的大小與範圍，每一個微型服務擁有自己的執行

* 本研究接受科技部編號：MOST 104-2218-E-035 -016 研究計畫經費補助

程序，彼此間透過輕量化的通訊機制溝通，且針對微型服務本身的特性可採用不同程式語言實作與使用合適的底層資料庫。

不同於 Monolithic 架構將整個軟體系統視為單一個執行單元，Microservices 架構是以軟體元件為中心，每一個軟體元件包裝成微型服務(microservice)可獨立運作，並可獨立部署於雲端的環境中(如圖二所示)，若微型服務需要底層資料庫支援，可以根據需求選擇合適的專屬資料庫，例如推薦或搜尋服務就可以採用 NoSQL 資料庫。此外當發現到某個微型服務負載增加時 Microservices 架構不會為整體應用程式建立複本，而是只針對該微型服務作水平性的擴展，在負載管理上更具彈性。



圖二 基於 Microservices 架構的電子商務網站規劃

相較於 Monolithic 架構，Microservices 架構更合適於現今的敏捷式軟體開發流程。當某個微型服務需要更新時，只需針對該服務作重新的建構、測試與部署，可縮短產品或服務的遞送時程與加速功能更新的頻率。除此之外，由於每一個微型服務都有明確的定義與界限，對於新進員工而言，可以縮短理解與開發時間，進而提升軟體開發產能(Productivity)。而在容錯方面，微型服務可增加錯誤隔離性(Fault Isolation)，例如當服務發生記憶體洩漏(Memory Leak)時，問題只會存在該服務中而不會蔓延影響到整體系統效能。

儘管 Microservices 架構擁有上述的優點，然而 Microservices 架構是以微型服務為單位部署在分散的雲端環境中，雖然具有彈性，另一方面也意味著微型服務的監控與管理更為複雜。隨著客戶端請求量的改變、微型服務數量的成長以及雲端環境的變動，提供微型服務應用程式之供應者(Application Provider)，希望有一機制可以報告從最小的微型服務到整體應用程式的服務品質(Quality of Service, QoS)之狀態，當應用程式服務品質未達設定服務品質標準時，此機制可以即時地警告應用程式供應者並可迅速地找出是那些微型服務影響服務品質下降。

有鑑於上述微型服務應用程式供應者之需求，本研究發展一個整合軟體代理人技術與 ElasticSearch、Logstash 和 Kibana (ELK) Stack[7]的服務監測機制可用於 Microservices 架構下的應用程式之營運分析與管理。不同於其他雲端服務監測機制，其主要考量為系統層級的監測資料，如 CPU 負載和記憶體的使用等。我們所提出的監測機制主要專注於應用程式層級的行為，如客戶端請求數與可獲得性(Availability)等。此機制所提供的主要功能如下：(1)針對不同型態的微型服務提供相對應的監測機制以收集分散在雲端伺服器上的服務監測資料、(2)允許使用者指定的 KPI (Key Performance Indicator)，KPI 值此由基本度量指標(metric)資料計算出以獲得聚合後的分析結果、(3)提供一個視覺化儀表板(Dashboard)，應用程式供應者可以從各種分析圖表中獲得的洞察。

本篇論文章節安排如下，第二節是相關研究介紹，第三節說明此監測機制的設計架構，第四節將介紹如何應用此監測機制於微型服務應用程式上，最後第五節為結論。

二、相關研究

目前已有應用代理人技術於雲端服務監控方面相關研究。Hasselmeyer *et al.*[8]針對虛擬的資料中心提出了可以支援多客戶的雲端服務監控方法，文獻中作者發展了基於代理人技術的雲端服務監控機制，利用代理人收集監測資料，再將收集到的資料送至統一的訊息匯流排(Message Bus)做進一步的過濾與聚合，最後將處理過的監測資料儲存在資料庫中，客戶可透過開發的監控程式存取資料庫查看監控結果。

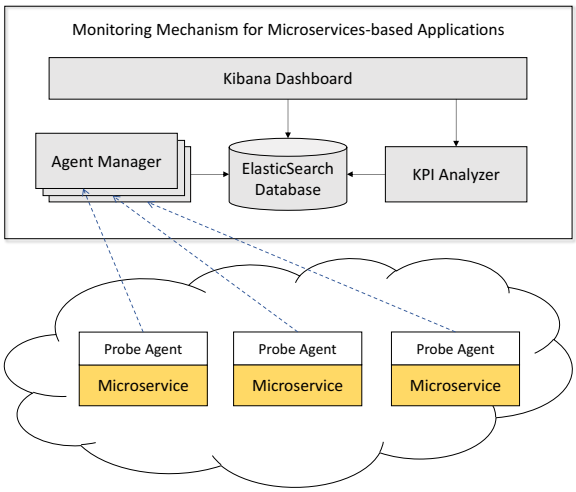
Kutare *et al.*[9]提出了一個名為 Monalytics 的監測與分析系統。針對雲端資料中心作者將監控系統架構規劃成一個計算通訊拓譜圖(Computational Communication Graph)，根據資料中心的監控分析需求、當下狀態與負載，計算通訊拓譜圖可被彈性的重新建構，此拓譜圖設計與實作可以根據資料中心的基礎架構設定為集中式、階層式或點對點式等結構。而代理人執行在每一個節點上的各層級如虛擬層、系統層與應用層等，用以擷取與處理監測的資料。

Emeakaroha *et al.*[10]提出了 LoM2HiS 框架用以偵測 SLA 協議在未來時間將會違反的危脅，並通知 Enactor 元件動作以避免違反 SLA 協議的發生，此外 LoM2HiS 框架也管理低階資源度量(Low-level resource Metrics)與高階 SLA 間的對應關係。在此框架中，作者說明了二種監測機制：(1)以代理人實作的 Host Monitor 用以監測硬體與網路資源並收集原生度量(Row Metrics)資料；(2) Run-time Monitor 用以持續的監測客戶應用程式的狀態與效能，其將 Host Monitor 所收集的原生度量資料轉換成高階的 SLA 參數，並偵測所分析的 SLA 參數值是否將違反 SLA 協議，若是則送出通知訊息。

此外，近二年來 Linux Container 技術的成熟，已成為輕量化虛擬機器的解決方案。Linux Container 是一種作業系統層級的虛擬化技術，不需要 Hypervisor 這個軟體層，它將應用軟體系統打包成一個軟體容器（Container），內含應用軟體本身的程式，以及所需要的作業系統核心和函式庫。透過統一的命名空間和共用 API 來分配不同軟體容器的可用硬體資源，創造出應用程式的獨立沙箱(Sandbox)執行環境，使得使用者可以容易地建立與管理虛擬與隔離的軟體作業環境。

Docker[11]是架構在 Linux Container 上的虛擬化開源軟體，Docker 對 Linux Container 作進一步的封裝，提供介面讓使用者可以容易地與安全地操作與管理容器。Docker 中容器是執行映像檔的結果，在此 Docker 強化了映像檔的觀念，如同版本控管系統一樣，Docker 增加了資料層級在映像檔上，每次修改容器的內容後可以交付新的資料層級加到原有的映像檔上。

此外，Docker 還提供了 Docker Hub[12]讓使用者可以分享客製化的映像檔，另一方面使用者也可以搜尋公開發佈的映像檔並下載到本地端的 Docker 環境上執行。藉由 Docker Hub 的分享與合作機制讓 Docker 成為現今最受歡迎的 Container 工具。



圖三 監測機制之系統架構

三、監測機制之設計

相較於 Monolithic 應用程式，其中大部分的軟體元件皆部署在單一個伺服器上。而微型服務在本質上是分佈於雲端環境上運作執行，因此如何有效地監控這些具有分散特性的微型服務以了解其營運狀態已經成為一個重要的挑戰。另一方面，雖然許多研究提出了相關的服務監測機制，這些研究主要還是集中在系統層級的狀態資料，例如 CPU 負載與網路流量等，然而較少探討應用程式層級的狀態資料，例如客戶端請求數、每次服務請求之成本、錯誤發生的頻率等。對於應用程式供應者而言，應用程式層級的狀態資料相較於系統層級的資料更具有價值，尤其是在做決策的時候。因此在本研究中，我們提出了針對微型服務軟體架構之應用程式的監測機制，藉由此機制應用程式供應者可以容易地了解其所提供的應用程式之整體營運狀況。

圖三為本研究所提出的監測機制之系統架構，其主要由 Probe Agent、Agent Manager、ElasticSearch、KPI Analyzer 與 Kibana Dashboard 等五個模組所構成。下列為我們對系統架構中每一個軟體模組的詳細說明：

Probe Agent：針對每一個微型服務都會有相對應的 Probe Agent 用以持續地監測微型服務所生成的監測資料。同時 Probe Agent 解析與過濾所收集的 log 並將其轉換成有效的度量資料，最後將度量資料傳送給 Agent Manager。

Agent Manager：當 Probe Agent 加入或離開監測機制平台時，Agent Manager 負責管理 Probe Agent 的註冊。同時 Agent Manager 也負責接收由 Probe Agent 所收集的監測度量資料並接著轉存至 ElasticSearch 資料庫中。為了減輕單一 Agent Manager 的工作負載，所有的 Probe Agent 將根據其地域或功能等屬性作分群，針對每一群 Probe Agent，皆存在一 Agent Manager 負責與該群的 Probe Agent 溝通。

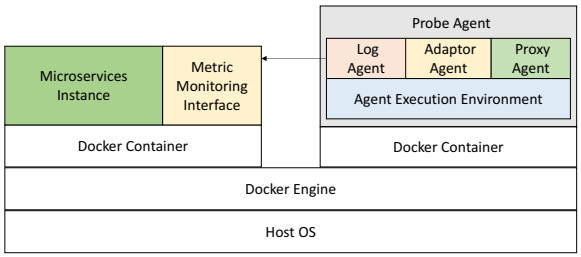
ElasticSearch：其作為底層分散式資料庫，負責管理 Probe Agent 所收集的監測度量資料並提供給客戶端程式查詢、聚合與分析等應用程式介面。

表一 KPI 計算方程式範例

KPI	方程式
Availability	$A = 1 - \frac{downtime}{uptime}$
Response Time	$R_{total} = \frac{Packetsize}{availablebandwidthin - inbytes} + \frac{Packetsize}{availablebandwidthout - outbytes}$
Reliability	$R = \frac{successful\ responses}{requests}$
Cost/Request	$Cost/Req = \frac{service_{cost}}{service_{requests}}$

KPI Analyzer：Probe Agent 主要從每個微型服務收集基本的監測度量資料，例如客戶端請求數等。然而對於應用程式供應者而言，對高階的 KPI 分析數據更具有意義。高階的 KPI 分析數據是由基本的監測度量資料計算而來(表一)，例如可靠度其值是由成功的回覆除以所有的客戶端請求數計算而來。KPI Analyzer 可以讓使用者指定高階的 KPI 方程式以作進一步地資料分析。

Kibana Dashboard：為一個基於 Kibana 所開發的 Web 使用者介面，藉由此介面資料分析師可以探索從微型服務所收集的監測度量資料並根據資料的特性制定各種分析圖表，而一般使用者可以透過分析圖表查看從大至應用程式到小至每一個微型服務的營運狀況。



圖四 Probe Agent 的軟體堆疊

由於 Probe Agent 的開發為本研究的重點，其詳細說明如下。Probe Agent 本身為微型服務被包裝與部署於 Docker 容器中。圖四說明了 Probe Agent 的軟體堆疊以及 Probe Agent 與其他微型服務實體(instance)彼此間的關係，Probe Agent 可與微型服務實體部署在相同的實體或虛擬機器上以收集微型服務的監測度量資料，而我們擴充先前行動代理人的研究成果[13]發展針對微型服務軟體架構下的 Probe Agent，其軟體模組之設計主要基於 OSGi 平台技術[14]。Agent Execution Environment 一個可以提供安裝、啟動、執行、停止與移除 Probe Agent 的代理人執行環境，藉此可以動態地管理在其上運作的所有 Probe Agent。在此研究中，我們針對不同的微型服務類型，將 Probe Agent 進一步地細分成三種不同型態的代理人：

Log Agent：為本研究中最主要的智慧型代理人，負責收集應用程式供應者自行開發的微型服務之監測度量資料。由於 Docker Container 是目前最普遍被採用來實作微型服務之技術。本研究假設應用程式供應者自行開發的微型服務都是部署在 Docker Container 上執行。誠如上述，本研究之重點在收集與分析應用程式層級的監測資料，微型服務所產生的 Log 資料揭露有用的訊息，透過有效的 Log 分析可以幫助使用者了解應用程式的營運情形。為了能讓外界監測微型服務的狀態，我們定義了一個名為 Metrics Monitor Interface 的 Log 資料取得介面，每一個受監測的微型服務必須實作此應用程式介面讓外界了解如何取得 Log 資料。下列是以 JSON 格式為例的監測資料取得方式，外界可以透過此資訊了解 Catalog service 的 access log 是以 ASCII 的格式存放於/mnt/catalog/logs/access.log 中，而其生成方式為不斷地將新的狀態添加到檔案裡。

```
{
  "microservice_name": "Catalog Service",
  "url":"file:///mnt/catalog/logs/access.log",
  "description":"access log",
  "format":"ASCII",
  "generated_method": "append"
}
```

每一個 Log Agent 都整合了 ELK Stack 所提供的 LogStash 軟體模組，為了能有效地處理 Log，Log Agent 首先解析受監測微型服務所提供的 Metrics Monitor Interface 規格，而後收集其原始 Log 資料。接著整合於 Log Agent 的 LogStash 開始資料擷取的工作，包含根據指定的規則解析收集的 Log 資料、過濾不需要的資料與將其轉換成有意義的度量資料。最後 Log Agent 將處理完後的度量資料傳送到該群負責的 Agent Manager，Agent Manager 再將資料存於 ElasticSearch 分散式資料中作為分析使用。

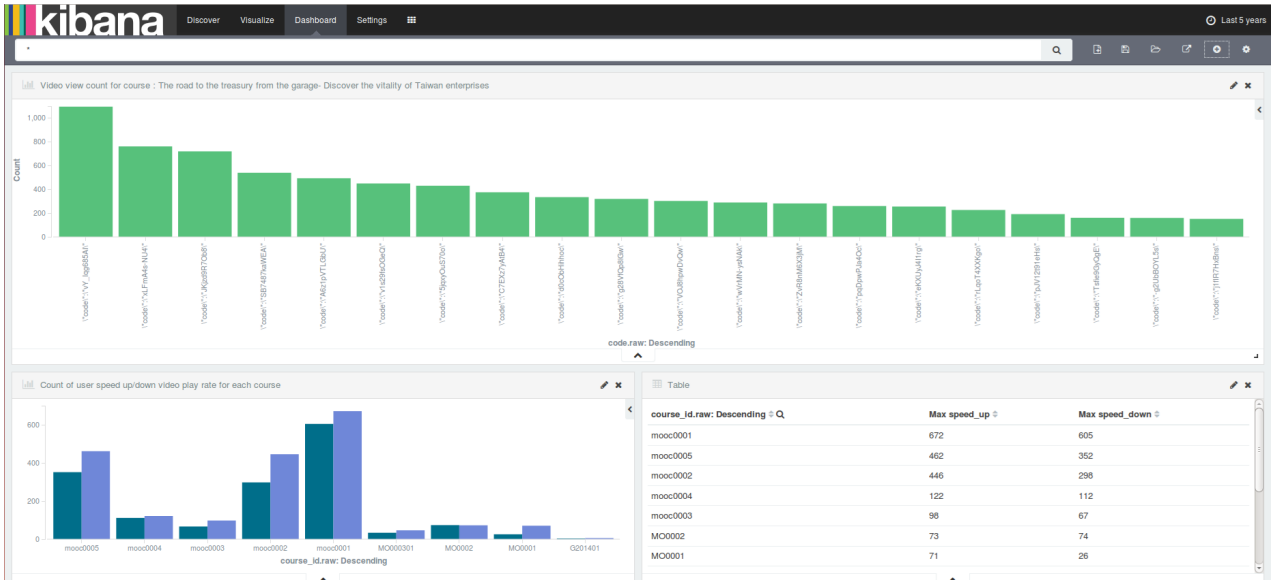
Adaptor Agent：其主要監測的對象為租用的微型服務且提供監測資料。如同 Log Agent 一樣，Adaptor Agent 也具有(1) 採掘能力：根據租用微型服務供應者所提供的監測介面取得監測資料；(2) 轉換能力：將收集到的監測資料解析、過濾與最後轉換成統一監測資料格式；(3)載入功能：將轉換後的監測資料傳送給 Agent Manager 儲存。

Proxy Agent：其主要監測的對象為租用的微型服務但沒有提供監測資料。為了能監測到此類的微型服務，Proxy Agent 採用代理(proxy)的方法，當應用程式叫用此類微型服務時，會先將叫用訊息傳送至此代理人，此代理人會代理應用程式呼叫租用的微型服務，在訊息往返同時將監測資訊紀錄下來，最後傳送給 Agent Manager 儲存。

四、應用展示

為了驗證此研究所提出針對微型服務應用程式之監測機制的可行性，我們將所提出的監測機制應用於監測與分析 OpenEdu [15] Web 應用程式之營運管理，OpenEdu 是一個建立於 Open edX[16] 開源學習平台的 MOOCs (Massive Open Online Courses)[17] 應用程式。OpenEdu 為線上學習平台，教師可以在其上提供具有影片、測驗與討論等輔助教材的課程，而學生可以不受時間與地點限制，在註冊後便能在平台上修習線上課程。由於 OpenEdu 在營運時無時無刻都會產生 Log，這些 Log 若能有效地收集與分析，對教師或平台管理者而言都是極具價值的資料。為此我們將 OpenEdu 包裝成微型服務並將其部署於 Docker 容器中，利用所發展的 Probe Agent 擷取 OpenEdu 的原始 Log 並在處理後存放於 ElasticSearch 資料庫中，最後利用 Kibana 儀表板將分析數據以圖表的方式呈現。

圖五為 Kibana 儀表板之截圖，其顯示 OpenEdu 平台上課程影片使用情形的分析結果。在此展示中，網頁上方區塊為 2014 年某一特定課程其每一個課程影片被觀看次數 KPI 的分析結果。教師透過檢視此圖表可以探討某些課程影片觀看次數較少的原因。網頁下方的柱狀圖與表格呈現學生在觀看 2014 年度每一課程的課程影片時加速與減速的次數。根據分析的結果，針對學生頻繁加速的影片，教師可以研究影片內容是否過於簡單導致無法吸引學生觀看。相對地，針對學生頻繁減速的影片，教師可以研究影片內容是否過於困難導致學生必須放慢速度學習。基於上述的觀察，教師可以持續地調整與改進課程影片的內容與品質。



圖五 Kibana 儀表板

五、結論

日前微型服務軟體架構已逐漸被大型資訊技術公司如 Netflex 與 Amazon 所採用。微型服務軟體架構已展現出眾多相較於傳統 Monolithic 軟體架構的優點，然而如何有效地管理分散的微型服務之營運仍是個重要的議題。因此在本研究中我們提出了一個結合軟體代理人與 ELK Stack 技術的監測機制，透過將原始 Log 資料轉換成有意義的分析數據，協助使用者了解應用程式的營運狀態。未來我們將擴展此機制加入警告與自動部署功能。警告功能用以提醒應用程式供應者當所監測到的度量值超過所指定的門檻。另一方面，當系統偵測到某一微型服務實體負載過重時，自動部署功能可以利用移動代理人自動地生成與部署微型服務複本實體以平衡工作負載。

參考文獻

[1] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*: Prentice Hall, 2003.

[2] Ken Schwaber, *Agile project management with Scrum*: Microsoft Press, 2004.

[3] Martin Fowler, *Patterns of Enterprise Application Architecture*: Addison Wesley, 2002.

[4] Lucas Krause, *Microservices: Patterns and Applications: Designing fine-grained services by applying patterns*: Lucas Krause, 2015.

[5] J. Lewis and M. Fowler, *Microservices http://martinfowler.com/articles/microservices .html*, 2015.

[6] V. Cardellini, M. Colajanni and S.Y. Philip, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, Vol. 3, No. 3, 1999.

[7] ELK stack, https://www.elastic.co/webinars/introduction-elk-stack.

[8] P. Hasselmeyer and N. d'Heureuse, "Towards holistic multi-tenant monitoring for virtual data centers," *Network Operations and Management Symposium Workshops*, 2010.

[9] M. Kutare , K. Schwan , G. Eisenhauer , V. Talwar , C. Wang and M. Wolf, "Monalytics - online monitoring and analytics for large scale data centers," *7th International Conference on Autonomic Computing*, 2010.

[10] V. C. Emeakaroha, M. A.S. Netto, R. N. Calheiros, I. Brandic, R. Buyya and C. A.F. De Rose, "Towards autonomic detection of SLA violations in Cloud infrastructures," *Future Generation Computer Systems*, Vol. 28, 2012.

[11] Docker, https://www.docker.com/

[12] Docker Hub, https://hub.docker.com/

[13] J. Lee, S. Lee, H. Chen and K. Hsu, "Itinerary-based Mobile Agent as a Basis for Distributed OSGi Services," *IEEE*

Transactions on Computers, Vol. 62, No. 10, 2013.
[14] OSGi Service Platform, release 3, OSG Alliance, 2003.
[15] OpenEdu, https://www.openedu.tw.
[16] Open edX, https://open.edx.org/.
[17] Ken Masters, "A brief guide to understanding MOOCs," *The Internet Journal of Medical Education*, Vol. 1, No. 2, 2011.