การเปรียบเทียบประสิทธิภาพระหว่างสถาปัตยกรรมแบบโมโนลิธ และไมโครเซอร์วิสโดยใช้ดอคเกอร์และคูเบอร์เนตส์

PERFORMANCE COMPARISON BETWEEN MONOLITH AND MICROSERVICES USING DOCKER AND KUBERNETES

นภวิชญ์ ทุมวงษ์

บัณฑิตวิทยาลัย มหาวิทยาลัยศรีนครินทรวิโรฒ

2562

การเปรียบเทียบประสิทธิภาพระหว่างสถาปัตยกรรมแบบโมโนลิธ และไมโครเซอร์วิสโดยใช้ดอคเกอร์และคูเบอร์เนตส์

นภวิชญ์ ทุมวงษ์

สารนิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต สาขาวิชาเทคโนโลยีสารสนเทศ
คณะวิทยาศาสตร์ มหาวิทยาลัยศรีนครินทรวิโรฒ
ปีการศึกษา 2562

PERFORMANCE COMPARISON BETWEEN MONOLITH AND MICROSERVICES USING DOCKER AND KUBERNETES

NAPAWIT TOOMWONG

A Master's Project Submitted in Partial Fulfillment of the Requirements

for the Degree of MASTER OF SCIENCE

(Information Technology)

Faculty of Science, Srinakharinwirot University

2019

สารนิพนธ์

เรื่อง

การเปรียบเทียบประสิทธิภาพระหว่างสถาปัตยกรรมแบบโมโนลิธ

และไมโครเซอร์วิสโดยใช้ดอคเกอร์และคูเบอร์เนตส์

ของ

นภวิชญ์ ทุมวงษ์

ได้รับอนุมัติจากบัณฑิตวิทยาลัยให้นับเป็นส่วนหนึ่งของการศึกษาตามหลักสูตร

ปริญญาวิทยาศาสตรมหาบัณฑิต สาขาวิชาเทคโนโลยีสารสนเทศ

ของมหาวิทยาลัยศรีนครินทรวิโรฒ

.....

(รองศาสตราจารย์ นายแพทย์ฉัตรชัย เอกปัญญาสกุล)

คณบดีบัณฑิตวิทยาลัย

.....

คณะกรรมการสอบปากเปล่าสารนิพนธ์

..... ที่ปรึกษาหลัก

(ผู้ช่วยศาสตราจารย์ ดร.วราภรณ์ วิยานนท์)

..... ประธาน

(ดร.สุทธิพงศ์ ธัชยพงษ์)

..... กรรมการ

(อาจารย์ ดร.วีรยุทธ เจริญเรืองกิจ)

ชื่อเรื่อง	การเปรียบเทียบประสิทธิภาพระหว่างสถาปัตยกรรมแบบโมโนลิธและไมโครเซอร์วิสโดยใช้ดอคเกอร์และคูเบอร์เนตส์
ผู้วิจัย	นภวิชญ์ ทุมวงษ์
ปริญญา	วิทยาศาสตรมหาบัณฑิต
ปีการศึกษา	2562
อาจารย์ที่ปรึกษา	ผู้ช่วยศาสตราจารย์ ดร. วราภรณ์ วิยานนท์

ปัจจุบันเทคโนโลยีเข้ามามีบทบาทในชีวิตประจำวันของเรามากขึ้น ประชากรส่วนใหญ่สามารถเข้าถึงเทคโนโลยีได้ง่ายไม่ว่าจะเป็นคอมพิวเตอร์หรือโทรศัพท์มือถือ ซึ่งปฏิเสธไม่ได้เลยว่าเทคโนโลยีนั้นเข้ามามีส่วนเกี่ยวข้องกับทุกกิจกรรมในการใช้ชีวิต แต่ในอีกมุมหนึ่งนั้นในโลกของการพัฒนาซอฟต์แวร์แล้วถือว่าได้รับผลกระทบเป็นอย่างมากจากการที่แอปพลิเคชันไม่สามารถรองรับปริมาณผู้ที่สามารถเข้าถึงเว็บไซต์หรือแอปพลิเคชันมือถือจำนวนมหาศาลเนื่องมาจากการออกแบบสถาปัตยกรรมที่ไม่รองรับในยุคที่เทคโนโลยีมีวิวัฒนาการอย่างรวดเร็ว ดังนั้นงานวิจัยนี้จึงมุ่งเน้นในการศึกษาและนำเสนอแนวคิดเกี่ยวกับการปรับปรุงโครงสร้างสถาปัตยกรรมเพื่อให้รองรับกับปริมาณผู้ใช้เพิ่มขึ้นโดยการจำลองพัฒนาแอปพลิเคชันโดยการนำซอฟต์แวร์ที่ถูกใช้กันอย่างแพร่หลายในองค์กรชั้นนำ เช่น ดอคเกอร์และคูเบอร์เนตส์โดยพัฒนาร่วมกับแนวคิดโครงสร้างสถาปัตยกรรมแบบโมโนลิธและไมโครเซอร์วิส โดยการทดสอบแบ่งออกเป็น 2 รูปแบบได้แก่ ทดสอบประสิทธิภาพของแอปพลิเคชันที่พัฒนาด้วยสถาปัตยกรรมแบบโมโนลิธด้วยดอคเกอร์และคูเบอร์เนตส์และประสิทธิภาพของแอปพลิเคชันที่พัฒนาด้วยสถาปัตยกรรมแบบไมโครเซอร์วิสด้วยดอคเกอร์และคูเบอร์เนตส์ ซึ่งในแต่ละรูปแบบถูกทดสอบด้วยการจำลองสร้างปริมาณการใช้งานและวัดการตอบสนองของแอปพลิเคชัน จากผลการวิจัยพบว่าสถาปัตยกรรมโมโนลิธและไมโครเซอร์วิสที่พัฒนาร่วมกับคูเบอร์เนตส์เพื่อทำให้แอปพลิเคชันรองรับการประมวลผลแบบขยายนั้นสามารถช่วยให้การตอบสนองของแอปพลิเคชันนั้นรวดเร็วยิ่งขึ้น โดยงานวิจัยในอนาคตมุ่งเน้นที่จะศึกษาและทดสอบกับแอปพลิเคชันที่มียูเซอร์ใช้งานจริงเพื่อพิสูจน์ว่าแนวคิดดังกล่าวสามารถนำไปประยุกต์ใช้ได้จริง

คำสำคัญ : โมโนลิธ, ไมโครเซอร์วิส, ดอคเกอร์, คูเบอร์เนตส์

Title	PERFORMANCE COMPARISON BETWEEN MONOLITH AND MICROSERVICES USING DOCKER AND KUBERNETES
Author	NAPAWIT TOOMWONG
Degree	MASTER OF SCIENCE
Academic Year	2019
Thesis Advisor	Assistant Professor Dr. Waraporn Viyanon

Nowadays, various types of technology play more of a role in our daily lives; for example, most people have easy access to technology via computers or mobile phones. It cannot be denied that technology is involved in all aspects in life. On the other hand, the world of software development can be affected by the inability of systems to accommodate the huge amount of people who access the website or application due to the design of architecture that no longer exists, in an age when technology has changed dramatically. In this paper, the performance comparison was tested and analyzed between a monolith and microservices using Docker and Kubernetes and by developing a simulation system based on these concepts. The performance comparison of web services in this study used the same scenarios with two different factors: using a monolith and microservices on Docker and Kubernetes. The results show that the monolith and microservices architecture developed with Kubernetes can reduce response times and increase throughput in the system. Moreover, it has explained the factors that made the system work in a more efficient way. Future studies may focus on studying and testing production systems to prove that these concepts can be applied.

Keyword : Monolith, Microservices, Docker, Kubernetes

กิตติกรรมประกาศ

สารนิพนธ์นี้สำเร็จลุล่วงได้ด้วยความอนุเคราะห์จาก ผศ.ดร.วราภรณ์ วิยานนท์ อาจารย์ที่ปรึกษาที่ให้คำปรึกษา คำแนะนำในการทำสารนิพนธ์ ตลอดจนสนับสนุนข้อมูลทางวิชาการและข้อมูลสำหรับทำสารนิพนธ์นี้

ขอขอบคุณคณะกรรมการสอบวิทยานิพนธ์ได้แก่ ดร.สุทธิพงศ์ ธัชยพงษ์ อ.ดร.วีรยุทธ เจริญเรืองกิจ และ ผศ.ดร.จันตรีประเสริฐผล ที่กรุณารับเป็นประธานและกรรมการสอบสารนิพนธ์และให้คำแนะนำ คำติชมสารนิพนธ์เล่มนี้ ซึ่งข้อติชมเหล่านั้นผู้จัดทำได้นำมาปรับปรุงสารนิพนธ์ฉบับนี้และเก็บไว้เป็นความรู้ในการศึกษาต่อไป

ขอขอบคุณอาจารย์และพนักงานประจำภาควิชาวิทยาการคอมพิวเตอร์ที่ให้ความรู้ คำแนะนำ และขอขอบพระคุณเจ้าหน้าที่ห้องธุรการประจำภาควิชาวิทยาการคอมพิวเตอร์ทุกท่านที่อำนวยความสะดวกด้านเอกสาร ตลอดจนคำปรึกษาในขั้นตอนดำเนินการจนสารนิพนธ์เล่มนี้เสร็จสมบูรณ์

นภาวิชญ์ ทุมวงษ์

สารบัญ

หน้า

สารบัญตาราง

หน้า

สารบัญรูปภาพ

หน้า

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ปัจจุบันเทคโนโลยีต่างๆ เข้ามามีบทบาทในชีวิตประจำวันของเรามากขึ้น ประชากรส่วนใหญ่สามารถเข้าถึงเทคโนโลยีได้ง่ายไม่ว่าจะเป็นคอมพิวเตอร์หรือโทรศัพท์มือถือ ซึ่งปฏิเสธไม่ได้เลยว่าเทคโนโลยีนั้นเข้ามามีส่วนเกี่ยวข้องกับทุกกิจกรรมในการใช้ชีวิต สืบเนื่องมาจากราคาต้นทุนการผลิตที่ถูกลงหรือการแข่งขันในตลาดของผู้ผลิต แต่ในอีกมุมหนึ่งนั้นในโลกของการพัฒนาซอฟต์แวร์แล้วถือว่าได้รับผลกระทบเป็นอย่างมากจากการที่แอปพลิเคชันไม่สามารถรองรับปริมาณผู้ที่สามารถเข้าถึงเว็บไซต์หรือแอปพลิเคชันจำนวนมหาศาล เนื่องมาจากการออกแบบสถาปัตยกรรมที่ไม่รอบรับในยุคที่เทคโนโลยีมีวิวัฒนาการอย่างรวดเร็ว

ในโลกดิจิทัลทุกธนาคารต้องมีการปรับตัวให้เข้ากับยุคดิจิทัลโดยการพัฒนาช่องทางอำนวยความสะดวกให้แก่ผู้ใช้งาน ซึ่งทุกคนสามารถใช้บริการต่างๆ ของธนาคารผ่านอินเทอร์เน็ตโดยไม่ต้องเดินทางไปยังสาขา แต่เมื่อย้อนกลับไปก่อนที่จะมีเทคโนโลยีทันสมัยเหมือนปัจจุบันนั้น ในด้านของแอปพลิเคชัน Core banking ของธนาคารต่างๆ ได้ถูกพัฒนามาเพื่อตอบโจทย์การใช้งานในยุคนั้น ซึ่งการทำธุรกรรมต่างๆ นั้นทำได้ผ่านช่องทางสาขาหรือตู้เอทีเอ็มเท่านั้น ซึ่งการที่ธนาคารจะปรับโครงสร้างของสถาปัตยกรรมจากเดิมให้เข้าสู่ยุคดิจิทัลนั้นมีต้นทุนมหาศาลและมีความเสี่ยงต่อภาพลักษณ์ในด้านความน่าเชื่อถือทำให้ผู้ดูแลแอปพลิเคชันเลือกที่จะแก้ไขปัญหาในทางตรงข้ามคือการเพิ่มทรัพยากรของแอปพลิเคชันคอมพิวเตอร์แทน แต่อย่างไรก็ตามก็ไม่สามารถแก้ไขปัญหาดังกล่าวในระยะยาวได้เนื่องจากแอปพลิเคชันคอมพิวเตอร์นั้นมีข้อจำกัดของตัวมันเอง ซึ่งผลกระทบคือทุกๆ สิ้นเดือนธนาคารหลายๆ แห่งในประเทศไทยจะไม่สามารถใช้งานได้เนื่องจากแอปพลิเคชันขัดข้อง จากการเปิดเผยสถิติจากธนาคารแห่งประเทศไทย ไม่เพียงแต่อุตสาหกรรมการเงินเท่านั้นที่ได้รับผลกระทบแอปพลิเคชันตลาดซื้อขายออนไลน์ก็ได้รับผลกระทบไปด้วยเช่นกัน ไม่ว่าจะเป็นแอปพลิเคชันที่ไม่สามารถรองรับผู้ใช้งานได้ในช่วงสิ้นเดือน หรือผู้ใช้งานไม่สามารถซื้อหรือขายของได้ในช่วงที่แอปพลิเคชันสารสนเทศของธนาคารขัดข้อง ซึ่งแน่นอนว่าจะส่งผลกระทบไปยังอุตสาหกรรมอื่นๆ ต่อไปเป็นวัฏจักร

ในปัจจุบันไม่สามารถปฏิเสธได้เลยว่าการออกแบบโครงสร้างสถาปัตยกรรมที่ดีนั้นไม่เพียงแต่ต้องคำนึงถึงเรื่องประสิทธิภาพของการใช้งานแล้ว แต่ยิ่งไปกว่านั้นการออกแบบโครงสร้างสถาปัตยกรรมที่ดีนั้นต้องตอบโจทย์ด้านการรองรับปริมาณของผู้ใช้ และสามารถปรับตัวได้ง่ายกับวิวัฒนาการที่เปลี่ยนแปลงไปอย่างรวดเร็วของเทคโนโลยี ซึ่งในงานวิจัยนี้ผู้วิจัยจะนำเสนอแนวคิด

สถาปัตยกรรมการออกแบบ Monolith และ Microservices โดยพัฒนาร่วมกับ Docker และ Kubernetes เพื่อแก้ไขปัญหาที่ได้กล่าวมาข้างต้น

1.2 วัตถุประสงค์ของการวิจัย

1.2.1. เพื่อเปรียบเทียบประสิทธิภาพการทำงานของสถาปัตยกรรมแบบ Monolith และ Microservices โดยใช้ Docker และ Kubernetes

1.2.2. เพื่อศึกษาแนวคิดการออกแบบซอฟต์แวร์ที่รองรับการประมวลผลแบบกระจาย

1.2.3. เพื่อศึกษาแนวคิดการออกแบบสถาปัตยกรรมให้เหมาะสมกับรูปแบบของธุรกิจและองค์กร

1.3 ประโยชน์ที่คาดว่าจะได้รับ

1.3.1. เพื่อให้ทราบถึงประสิทธิภาพของการทำงานของสถาปัตยกรรมที่พัฒนาด้วยแนวคิดแบบ Monolith และ Microservices โดยใช้ Docker และ Kubernetes

1.3.2. เพื่อให้ทราบถึงวิธีการพัฒนาสถาปัตยกรรมที่ออกแบบเพื่อรองรับการประมวลผลแบบกระจาย

1.3.3. เพื่อให้ทราบถึงแนวคิดของการออกแบบสถาปัตยกรรมให้เหมาะสมกับความต้องการทางธุรกิจ

1.4 แผนการดำเนินการวิจัย

1.4.1 ศึกษาทฤษฎีและงานวิจัยที่เกี่ยวข้อง

1.4.2 ศึกษาวิธีการใช้งานซอฟต์แวร์แบบต่างๆ

1.4.3 พัฒนาแอปพลิเคชันต้นแบบ

1.4.4 ทดลองและปรับปรุงขั้นตอนวิธีเพื่อให้ได้ประสิทธิภาพมากขึ้น

1.4.5 วิเคราะห์ประสิทธิภาพของสถาปัตยกรรมที่ทดลองเชิงเปรียบเทียบ

1.5 ขอบเขตของการวิจัย

งานวิจัยนี้ทำการเปรียบเทียบประสิทธิภาพการทำงานของแอปพลิเคชันสะสมแต้มเพื่อแลกสินค้าที่พัฒนาด้วยแนวคิดแบบ Monolith และ Microservices โดยมีขอบเขตของการวิจัยดังต่อไปนี้

1.5.1. จำลองแอปพลิเคชันสะสมแต้มซึ่งประกอบไปด้วย 2 สถานการณ์ ได้แก่

S1. การเรียกดูสินค้า

S2. การแลกสินค้า

1.5.2. ทดสอบการจำลองปริมาณผู้ใช้ในระบบในรูปแบบ Load testing เพื่อทดสอบ Quality of services (QoS) ของระบบ

1.5.3. ทดสอบด้วยจำนวนโหลด 300 Transaction/second จำนวน 30 Threads ในรูปแบบ Ramp-up ซึ่งมีการเพิ่ม Thread ทุกๆ 10 วินาที โดยแบ่งการทดสอบเปรียบเทียบประสิทธิภาพได้เป็น 4 แบบ โดยทำการทดสอบซ้ำจำนวน 3 ครั้งต่อการทดสอบ โดยการทดสอบมีดังต่อไปนี้

1) พัฒนาด้วยแนวคิดแบบ Monolith โดยใช้ Docker

2) พัฒนาด้วยแนวคิดแบบ Monolith โดยใช้ Kubernetes

3) พัฒนาด้วยแนวคิดแบบ Microservices โดยใช้ Docker

4) พัฒนาด้วยแนวคิดแบบ Microservices โดยใช้ Kubernetes

1.6 ระยะเวลาดำเนินงานวิจัย

งานวิจัยนี้ใช้ระยะเวลาดำเนินการทั้งหมด 8 เดือน โดยแบ่งขั้นตอนและรายละเอียดตามตารางที่ 1 ดังนี้

ตาราง 1 ระยะเวลาการดำเนินงานวิจัย

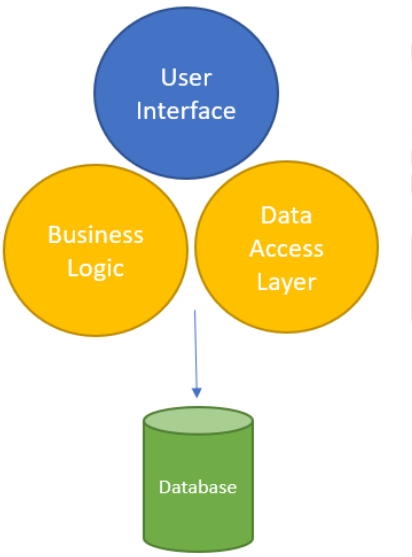
การดำเนินการวิจัย	ระยะเวลาศึกษา (เดือน)							
	1	2	3	4	5	6	7	8
ศึกษาทฤษฎีและงานวิจัยที่เกี่ยวข้อง								
ศึกษาวิธีการใช้งานซอฟต์แวร์แบบต่างๆ								
พัฒนาแอปพลิเคชันต้นแบบ								
ทดลองและปรับปรุงประสิทธิภาพ								
วิเคราะห์ประสิทธิภาพของสถาปัตยกรรมที่ทดลองเชิงเปรียบเทียบ								

บทที่ 2
ทบทวนวรรณกรรม

บทนี้จะกล่าวถึงสถาปัตยกรรมการออกแบบของ Monolith และ Microservices เพื่อให้เห็นถึงคุณลักษณะและแนวคิดของการออกแบบสถาปัตยกรรม รวมถึงส่วนประกอบที่สำคัญของ Docker และ Kubernetes ที่จะช่วยเพิ่มประสิทธิภาพและความคงทนให้กับแอปพลิเคชัน

2.1 Monolith

Monolith เป็นแนวคิดที่รวบรวมทุกฟังก์ชันการใช้งานและ Business logic ไว้ในแอปพลิเคชันชุดเดียวกัน ซึ่งข้อดีของแนวคิดนี้คือง่ายต่อการพัฒนาแอปพลิเคชันและใช้เวลาไม่นาน และข้อจำกัดของแนวคิดนี้คือการพัฒนาแอปพลิเคชันสถาปัตยกรรมที่ความผูกพันกันสูงซึ่งส่งผลให้การพัฒนาหรือแก้ไขบางส่วนของแอปพลิเคชันนั้นอาจส่งผลกระทบต่อทั้งแอปพลิเคชัน ยิ่งไปกว่านั้นการขยายแอปพลิเคชันนั้นไม่สามารถรองรับการขยายแอปพลิเคชันเฉพาะส่วนได้ดังภาพประกอบ 1 (Newman, 2015)

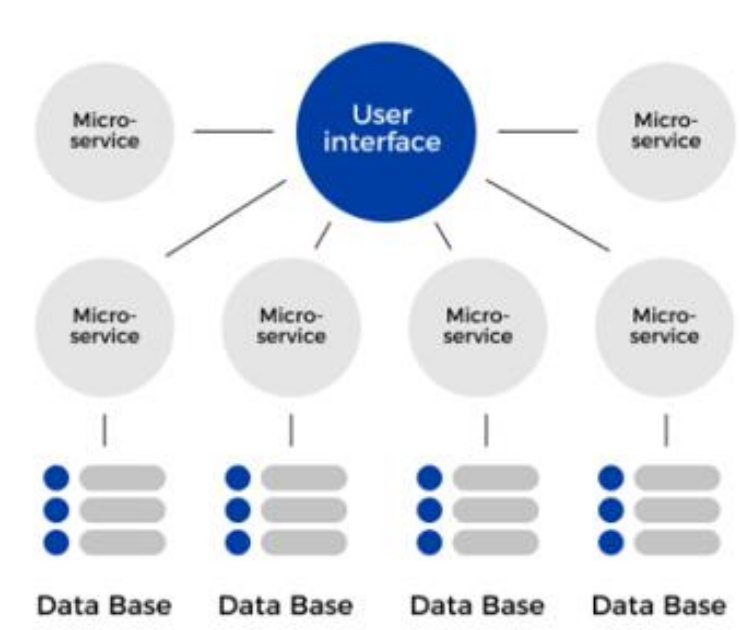


ภาพประกอบ 1 สถาปัตยกรรมแบบ Monolith

ที่มา: Mazur, O. (2020). Retrieved from Monolithic architecture vs microservices. Retrieved from https://divante.com/blog/monolithic-architecture-vs- Microservices /

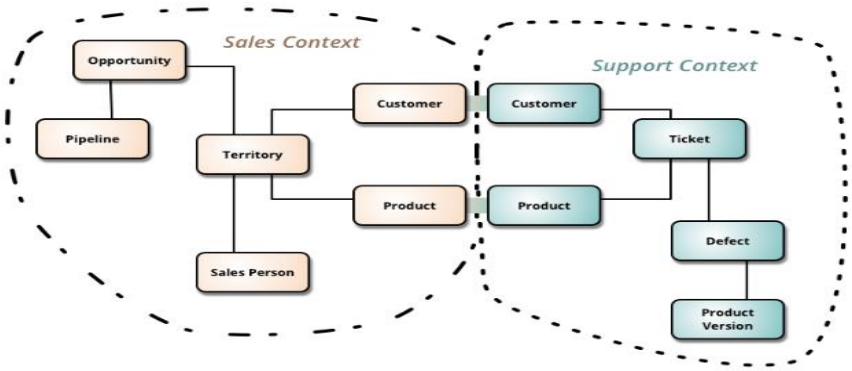
2.2 Microservices

Microservices เกิดจากแนวคิดการออกแบบสถาปัตยกรรมให้มีขนาดเล็กและมีความเป็นอิสระต่อกันโดยแต่ละเซอร์วิสสามารถทำงานร่วมกันได้ ซึ่งการแบ่งเซอร์วิสนั้นต้องมีการกำหนด Bounded context ให้ชัดเจน ซึ่งการออกแบบที่ดีนั้นแต่ละเซอร์วิสนั้นจะต้องมีความผูกพันกันน้อยที่สุด(Loose coupling) และสามารถทำงานด้วยตัวของมันเองได้โดยไม่ต้องอาศัยเซอร์วิสอื่น (High cohesion) ดังภาพประกอบ 2 โดยวิธีการแบ่งเซอร์วิสต่างๆ นั้นจะใช้เทคนิคการกำหนด Bounded context โดยการจำกัดขอบเขตของแบบแผนธุรกิจและความต้องการทางธุรกิจให้ออกมาอยู่ในรูปแบบของ Context ดังภาพประกอบที่ 3 ซึ่งจะช่วยให้การออกแบบและพัฒนาเซอร์วิสทำได้ออกมาชัดและง่ายขึ้น เมื่อสามารถกำหนด Bounded context ของแบบแผนธุรกิจทั้งหมดได้แล้ว ในส่วนของการพัฒนานั้นนักพัฒนาสามารถเลือกใช้ภาษาหรือเทคโนโลยีที่เหมาะสมกับความต้องการทางธุรกิจได้ (Newman, 2015)



ภาพประกอบ 2 สถาปัตยกรรมแบบ Microservices

ที่มา: Mazur, O. (2020). Retrieved from Monolithic architecture vs microservices. Retrieved from https://divante.com/blog/monolithic-architecture-vs- Microservices /



ภาพประกอบ 3 การแบ่ง Domain ด้วยการกำหนด Bounded context

ที่มา : Fowler, M. (2014). Retrieved from Fowler, M. Retrieved from https://martinfowler.com/bliki/BoundedContext.html

การทำให้เซอร์วิสติดต่อกับเซอร์วิสอื่นๆ นั้นโดยทั่วไปจะทำการติดต่อกันผ่าน REST (Representational state transfer) หรือ Message queue ซึ่งขึ้นอยู่กับการใช้งาน ซึ่งทั้งสองวิธีนี้สามารถพัฒนาโดยใช้ภาษาได้อย่างหลากหลายและมีประสิทธิภาพที่ดี

โดยการพัฒนาสถาปัตยกรรมแบบ Microservices ด้วยวิธีการออกแบบด้วย Bounded context (Vernon, 2016) ที่ดีนั้นสถาปัตยกรรมที่ถูกพัฒนาขึ้นมาควรมีคุณสมบัติดังต่อไปนี้

1) **ความหลากหลายของเทคโนโลยี** (technology heterogeneity)

ในแอปพลิเคชันสถาปัตยกรรมที่ประกอบด้วยเซอร์วิสที่มีขนาดเล็กทำงานร่วมกัน ควรเลือกใช้ซอฟต์แวร์หรือภาษาที่เหมาะสมกับเซอร์วิสนั้น เพื่อให้แอปพลิเคชันสามารถรองรับปริมาณการเข้ามาใช้งานได้อย่างเหมาะสมตาม Context นั้นๆ

2) **ความยืดหยุ่น** (resilience)

แอปพลิเคชันที่ถูกออกแบบมาด้วยแนวคิดแบบ Microservices เมื่อเซอร์วิสหนึ่งไม่สามารถใช้งานได้เซอร์วิสอื่นๆ ควรทำหน้าที่ต่อได้

3) **การขยายขนาด** (scaling)

ในแอปพลิเคชันสถาปัตยกรรมทั่วไปที่ถูกพัฒนาบนแนวคิดแบบ Monolith นั้น การขยายแอปพลิเคชันนั้นจะทำด้วยการขยายทั้งแอปพลิเคชันไปด้วยกัน แต่สถาปัตยกรรมที่ถูกพัฒนาด้วย Microservices นั้นสามารถขยายแอปพลิเคชันเฉพาะเซอร์วิสที่มีปริมาณผู้ใช้งานสูงได้

4) **ความง่ายต่อการส่งมอบ** (ease of deployment)

สถาปัตยกรรมแบบ Microservices สามารถพัฒนาและส่งมอบเฉพาะส่วนที่ถูกปรับปรุงหรือพัฒนาเพิ่มเติมได้

5) **การจัดการองค์กร** (organizational alignment)

การใช้แนวคิดแบบ Microservices ในองค์กรใหญ่สามารถแบ่งทีมพัฒนาออกเป็นทีมเล็กๆ ตามความถนัดของทีมนั้นๆ โดยไม่ต้องเรียนรู้ภาษาใหม่ๆ เพื่อนำพัฒนาแอปพลิเคชันที่ใช้เพียงภาษาเดียว

6) **การนำกลับมาใช่ใหม่** (composability)

การนำเซอร์วิสที่มีฟังก์ชันในการทำงานคล้ายกันนำกลับมาใช้ใหม่เพื่อจุดประสงค์อื่นได้ เช่น การนำเซอร์วิสที่มีฟังก์ชันยืนยันตัวตนจากแบบแผนธุรกิจหนึ่งไปใช้ในอีกแบบแผนธุรกิจหนึ่งโดยไม่จำเป็นต้องพัฒนาแอปพลิเคชันนั้นใหม่

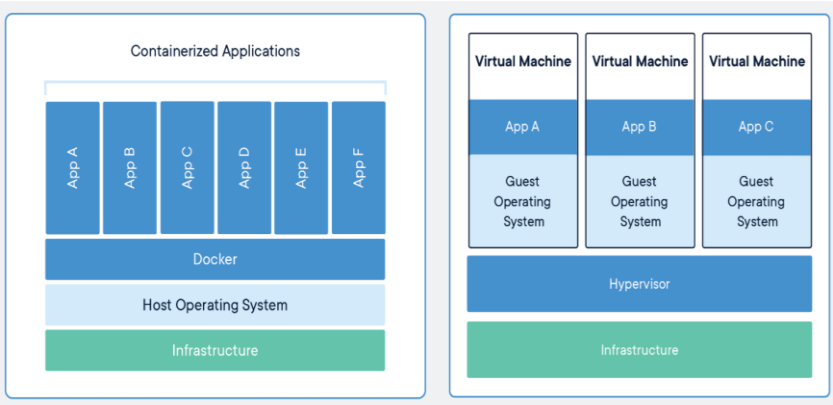
7) **ความง่ายต่อการแทนที่** (optimizing for replaceability)

การเปลี่ยนแปลงในบางเซอร์วิสนั้นต้องทำได้ง่ายและไม่กระทบต่อเซอร์วิสอื่น

2.3 Container

Container เป็นเทคนิคในการจำลองสภาพแวดล้อมหรือแอปพลิเคชันปฏิบัติการเสมือนขึ้นมาเพื่อจัดการกับซอฟต์แวร์ ซึ่งมีลักษณะคล้ายกับ Virtual machine ซึ่งซอฟต์แวร์ที่เป็นที่นิยมและได้รับการยอมรับอย่างกว้างขวางคือ Docker

Docker เป็นโอเพ่นซอร์สที่ถูกพัฒนาขึ้นเพื่อปรับปรุงประสิทธิภาพจากการใช้ Virtual machine แบบเดิมซึ่งใช้ทรัพยากรน้อยกว่าการใช้ Virtual machine ซึ่ง Docker สามารถติดตั้ง Container ได้มากกว่าหนึ่ง Container บนแอปพลิเคชันที่มี Operating System เดียวได้ซึ่งแตกต่างจาก Virtual Machine ที่ต้องติดตั้งบน Operating System ของตัวเองดังภาพประกอบ 4 ยิ่งไปกว่านั้นผู้ใช้สามารถจัดการกับ Container ได้อย่างง่ายผ่าน Docker engine ที่เป็นเครื่องมือสำหรับสร้าง (Build), ส่งมอบ (Ship), ใช้งาน (Run) ซึ่ง Docker ถูกนำมาเป็นส่วนหนึ่งของการพัฒนาระบบแบบสถาปัตยกรรมที่มีแนวคิดแบบ Microservices (Docker, 2019)



ภาพประกอบ 4 โครงสร้างของ Docker container และ Virtual machine

ที่ ม า : Fong, J. (2018). *Containers versus Virtual Machines*. Retrieved from https://www.docker.com/blog/containers-replacing-virtual-machines/

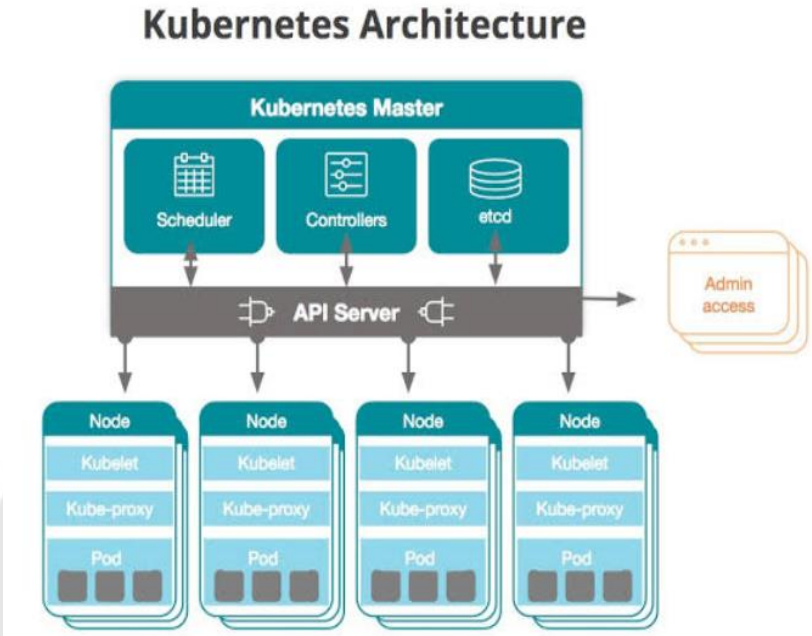
Docker นั้นถูกใช้ในหลายๆ บริษัทเนื่องจากมีข้อดีหลายประการ เช่น การใช้ทรัพยากรที่น้อย ติดตั้งซอฟต์แวร์หรือแอปพลิเคชันปฏิบัติการได้อย่างหลากหลายและอิสระต่อกัน

2.4 Kubernetes

Kubernetes เป็นแอปพลิเคชันโอเพ่นซอร์สที่ช่วยในการจัดการปรับขนาดและจัดการกับ Container โดยออกแบบมาให้ทำงานร่วมกับ Container โดย Kubernetes เป็นซอฟต์แวร์ที่ช่วยเพิ่มความทนทานให้แก่แอปพลิเคชันโดยใช้คุณสมบัติที่ Kubernetes มี เช่น การทำให้ระบบกลับมามีสภาพปกติ (Self-healing) หรือ Horizontal scaling เป็นต้น ซึ่งจากคุณสมบัติที่กล่าวมาข้างต้น Kubernetes สามารถช่วยเพิ่มหรือขยาย Container ที่เกิดจากการใช้งานปริมาณและคืนทรัพยากรอย่างอัตโนมัติ โดยประโยชน์ของ Kubernetes นั้นมีจุดเด่นดังต่อไปนี้ (Hightower, Burns, & Beda, 2017; Kubernetes, 2019)

1. ช่วยให้ใช้ทรัพยากรที่มีจำกัดของฮาร์ดแวร์ทำงานได้อย่างเต็มประสิทธิ์ภาพ
2. เพิ่มลดหรือขยายทรัพยากรของ Container ได้อย่างอัตโนมัติ
3. ช่วยให้ Cluster ทำงานอย่างมีประสิทธิภาพ
4. แอปพลิเคชันสามารถจัดการกับความผิดปกติในเบื้องต้นได้
5. มีซอฟต์แวร์สำหรับการตรวจสอบทรัพยากรของแอปพลิเคชัน

โดยปกติแล้วในแอปพลิเคชันทั่วไปที่ใช้ Kubernetes จะประกอบไปด้วย Master และ Nodes โดย Master ทำหน้าที่ออกคำสั่งให้ Nodes ทำงานตามคำสั่งและ Nodes ทำหน้าที่รองรับการทำงานของ Container โดย Kubernetes architecture ประกอบไปด้วยส่วนต่างๆ ดังภาพประกอบ 5



ภาพประกอบ 5 ตัวอย่าง Kubernetes cluster

ที่มา : Gerrard, A. (2019). *a Kubernetes cluster*. Retrieved from https://blog.newrelic.com/engineering/what-is-kubernetes/

จากภาพประกอบที่ 5 แสดงให้เห็นว่า Kubernetes มีลักษณะการทำงานแบบ Cluster ที่ประกอบไปด้วย Master และ Node จำนวนมากที่ทำงานร่วมกัน โดย Kubernetes cluster มีส่วนประกอบดังต่อไปนี้

1) Master

Master มีหน้าที่ควบคุมการทำงานของ Node ใน Cluster ซึ่งทุกกิจกรรมที่จะเกิดขึ้น จะถูกสั่งงานผ่าน Master

2) Node

Node เปรียบเสมือนเครื่องคอมพิวเตอร์หรือ Virtual Machine ที่ทำหน้าที่จัดเก็บ Container โดยติดต่อกับ Master ผ่านเครื่องมือที่ชื่อว่า Kubelet

3) Pod

Pod คือหน่วยที่เล็กที่สุดของ Cluster ซึ่ง Pod ทำหน้าที่บรรจุ Container และ Resource ที่ใช้ในการติดตั้ง เช่น การกำหนดการเชื่อมต่อเครือข่าย (Network configuration), การกำหนดการจัดเก็บข้อมูล (Storage configuration) และการกำหนดค่าอื่นๆ ที่จำเป็น

4) Kube-proxy

Kube-proxy คือส่วนที่ใช้ในการจัดการกับ Network ใน Cluster โดยมีหน้าที่มอบ Internet protocol ให้กับ Node ภายใน Cluster

งานวิจัยนี้มีการใช้ฟีเจอร์ที่สำคัญของ Kubernetes คือ Horizontal pod autoscaling สำหรับการเพิ่มและขยายจำนวน Pod ตามปริมาณการใช้งาน โดย Horizontal pod autoscaling จะทำการตรวจสอบการใช้งาน CPU ว่ามีค่าสูงเกินกว่าค่าที่ตั้งไว้หรือไม่และจะทำการเพิ่มลดโดยอัตโนมัติ (Zhao et al., 2019) ซึ่งวิธีการคำนวณการจัดการเพิ่มลดและขยายแสดงดังสมการ 1

$$N = currentReplicas * \left(\frac{currentMetricValue}{desireMetricValue}\right) \quad (1)$$

โดยที่

N	คือ จำนวนของ Pod ที่จะถูกขยายเพิ่มขึ้น
currentReplicasValue	คือ จำนวน Pod ปัจจุบัน
currentMetricVallue	คือ ค่าของ CPU ที่แอปพลิเคชันใช้ในขณะทำงาน
desireMetricValue	คือ ค่าของ CPU ที่ต้องการให้แอปพลิเคชันขยายตัว

โดยปกติแล้ว Kubernetes จะมีหน่วยที่ใช้ในการวัดทรัพยากรที่ใช้เรียกว่า millicores โดยที่ 1 Core CPU จะมีค่าเท่ากับ 1,000 millicores จากสมการที่ 1

ตัวอย่างการคำนวณค่า N สมมุติให้ในขณะที่แอปพลิเคชันกำลังทำงาน Pod มีการใช้ทรัพยากร (currentMetricValue) 280 millicores โดยที่ผู้ใช้ตั้งค่าให้ HPA ให้ทำการขยายตัวเมื่อ CPU ถูกใช้งานมากกว่า 50% โดยที่ Pod มีทรัพยากรจำกัดที่ 500 millicores ดังนั้น desireMetricValue จะมีค่าเท่ากับ 250 millicores เมื่อ currentMetricValue มีค่ามากกว่า desireMetricValue แล้ว Master จะทำการขยาย Pod ออกจำนวน 2 อินสแตนท์ เพื่อรักษาการใช้ CPU ไม่ให้เกิน 50% ตามที่ผู้ใช้กำหนด

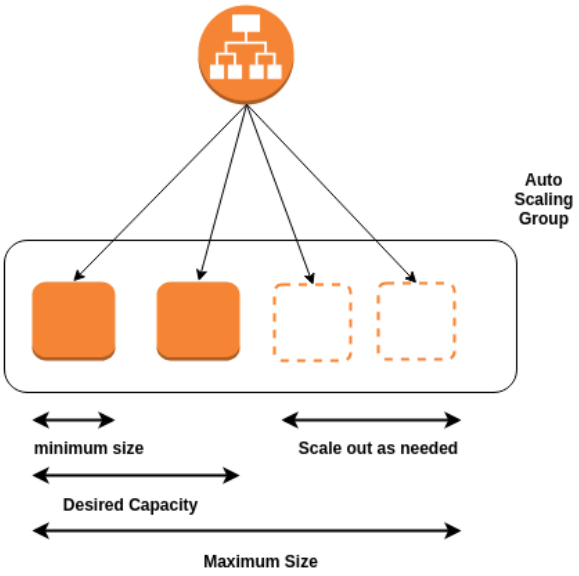
2.5 Cloud-native application (google cloud)

Cloud-native application คือแนวคิดหนึ่งที่ใช้ในการออกแบบพัฒนาแอปพลิเคชัน สถาปัตยกรรมที่สามารถรองรับการเติบโตของธุรกิจโดยใช้เทคโนโลยีที่สามารถสร้างและปรับเปลี่ยนตามยุคสมัยได้ในทุกสภาพแวดล้อม เช่น Private cloud, Public cloud และ Hybrid cloud ซึ่งแต่ละที่จะมีแนวคิดที่แตกต่างกันออกไปแต่เพื่อจุดประสงค์เดียวกันคือแอปพลิเคชันนั้นจะต้องรองรับความหยืดหยุ่นในด้านการจัดการทรัพยากร การสื่อสารที่ไม่จดจำรูปแบบการเข้าใช้งานครั้งก่อน ค่าใช้จ่ายและการเพิ่มลดขยายได้อย่างอัตโนมัติ

องค์ประกอบของ Cloud native มุ่งเน้นไปที่การปรับตัวและลดต้นทุนการผลิตโดยแก้ไขปัญหาดังต่อไปนี้ Loose coupling, High cohesion, Stateless และ Resilience เป็นต้น (Cloud, 2019)

2.6 Autoscaling

Autoscaling เป็นวิธีการที่ใช้ในการประมวลผลแบบคลาวด์เพื่อปรับแต่งทรัพยากรของแอปพลิเคชันตามความต้องการเช่น การปรับขนาดของทรัพยากรโดยอัตโนมัติช่วยให้แอปพลิเคชันสามารถขยายหรือลดขนาดเพื่อให้ทรัพยากรเพียงพอสำหรับความต้องการของผู้ใช้งาน นอกจากนี้เมื่อแอปพลิเคชันเข้าสู่ภาวะปกติ แอปพลิเคชันจะทำการลดขนาดโดยอัตโนมัติ ซึ่งวิธีการนี้สามารถลดภาระในการตรวจสอบและดูแลแอปพลิเคชันที่ใช้ทรัพยากรบุคคล (Kleppmann, 2017) จากภาพประกอบ 6 ผู้ใช้ต้องกำหนดจำนวนทั้งหมดของอินสแตนท์ (Maximum size) และขนาดของอินสแตนท์เริ่มต้น (Desired capacity) เมื่อแอปพลิเคชันได้รับปริมาณการใช้งานที่สูงขึ้นระบบจะทำการเพิ่มปริมาณตามอินสแตนท์ที่ผู้ใช้ได้กำหนดโดยอัตโนมัติ ซึ่งผู้พัฒนาจำเป็นต้องเข้าใจพฤติกรรมการใช้งานของลูกค้าและปรับปรุงการกำหนดค่าต่างๆ ของ Autoscaling ให้เข้ากับสถานการณ์อยู่เสมอ



ภาพประกอบ 6 Autoscaling model

ที่มา: Tudip (2018). Retrieved from Amazon Auto Scaling: Getting Started with AWS.Retrieved from https://tudip.com/blog-post/amazon-autoscaling/

2.7 งานวิจัยที่เกี่ยวข้อง

การทบทวนวรรณกรรมของงานวิจัยนี้ได้ทำการศึกษางานวิจัยที่เกี่ยวข้องกับสถาปัตยกรรมแบบ Microservices และ Monolith มีรายละเอียดดังต่อไปนี้

(1) บทความวิจัยเรื่อง Evaluating the Monolithic and the Microservices Architecture Pattern to Deploy Web Applications in the Cloud (Villamizar et al., 2015) งานวิจัยนี้ได้ทำการเปรียบเทียบประสิทธิภาพและค่าใช้จ่ายของการใช้งานบน Amazon web services ของสถาปัตยกรรมแบบ Monolith และ Microservices โดยการทดสอบแอปพลิเคชันจากการใช้งานจริงโดยสถาปัตยกรรมแบบ Monolith ประกอบไปด้วย Web application ที่พัฒนาด้วย Play 2.2.2, Scala 2.10.2 และ Java 1.7.0 โดยใช้ฐานข้อมูล PostgreSql 9.3.6 และส่วน Front-end application พัฒนาด้วย Angular.js 1.3.14 (HTML5, CSS และ JQuery) และสถาปัตยกรรมแบบ Microservices ประกอบไปด้วย 2 อินสแตนท์ที่แยกออกเป็น Web application และ Front-end application ที่ใช้ซอฟต์แวร์เสมือนกับสถาปัตยกรรมแบบ Monolith

ซึ่งทั้งหมดทำการจำลองแอปพลิเคชันสินเชื่อ โดยแบ่งออกเป็น 2 เซอร์วิสคือ การสร้างสินเชื่อ การเรียกดูสินเชื่อที่เคยซื้อไปแล้ว

โดยผลลัพธ์ที่ได้พบว่าค่าใช้จ่ายของแอปพลิเคชันสถาปัตยกรรมที่ติดตั้ง Microservices บน Amazon web services สามารถลดค่าใช้จ่ายได้มากถึง 17% และ Response time ของสถาปัตยกรรมแบบ Monolith นั้นมีค่าเฉลี่ยที่ดีกว่าสถาปัตยกรรมแบบ microservices

(2) บทความวิจัยเรื่อง Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservices Architectures งานวิจัยนี้ได้ทำการเปรียบเทียบประสิทธิภาพและค่าใช้จ่ายของการใช้งานบน Amazon web services ของสถาปัตยกรรมแบบ Monolith ,Microservices และ Amazon lambda โดยสถาปัตยกรรมแบบ Microservices และ Monolith ถูกพัฒนาด้วย Play และ Jax-RS และ Amazon lambda ถูกพัฒนาด้วย Node.js ซึ่งทั้งหมดทำการจำลองแอปพลิเคชันสินเชื่อ โดยแบ่งออกเป็น 2 เซอร์วิสคือ การสร้างสินเชื่อ การเรียกดูสินเชื่อที่เคยซื้อไปแล้ว (Villamizar et al., 2016) โดยผลลัพธ์ที่ได้จากการจำลองเรียกข้อมูลจากแอปพลิเคชันสินเชื่อทั้งหมด ได้ผลลัพธ์ดังต่อไปนี้

1. สถาปัตยกรรมแบบ Microservices ที่พัฒนาด้วย Play สามารถลดค่าใช้จ่ายได้ถึง 9.50%, 13.81% และ 13.42% เมื่อเปรียบเทียบกับสถาปัตยกรรมแบบ Monolith ที่พัฒนาด้วย Jax-RS

2. สถาปัตยกรรมแบบ Microservices ที่ถูกพัฒนาบน Amazon lambda สามารถลดค่าใช้จ่ายได้ถึง 50.43%, 55.69% และ 57.01% เมื่อเทียบกับสถาปัตยกรรมแบบ Microservices ที่พัฒนาด้วย Play และ 55.14%, 61.81% และ 62.78% เมื่อเปรียบเทียบกับ Monolith ที่พัฒนาด้วย Jax-RS ยิ่งไปกว่านั้นสามารถลดค่าใช้เพิ่มขึ้นอีก 62.61%, 77.08% และ 70.23% เมื่อเปรียบเทียบกับสถาปัตยกรรมแบบ Monolith ที่พัฒนาด้วย Play

3. สถาปัตยกรรมแบบ Microservices นั้นมี Response time ที่สูงกว่าสถาปัตยกรรมแบบอื่นๆ

(3) บทความวิจัยเรื่อง K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters งานวิจัยนี้ได้ทำการทดลองการวัดประสิทธิภาพของ Cassandra database โดยการสร้างจำนวนโหลดเข้าสู่แอปพลิเคชันที่จำลองขึ้นมาคือ K8-Scalar ที่พัฒนาขึ้นเพื่อจะทำการวัด Latency ของ Database ในขณะที่

Kubernetes Horizontal pod autoscaling ทำการเพิ่ม Cassandra database อินสแตนท์ โดยการทดลองนี้ใช้ OpenStack cloud โดยขนาดของคอมพิวเตอร์มีคุณสมบัติ 2.60 GHz Intel Xeon E5-2660 processors และ 128GB DDR3 โดยแต่ละ Virtual machine ที่ใช้ในการทำวิจัยคือ Ubuntu 16.04 4 CPU cores และ 8GB memory โดยให้แต่ละ Pod สามารถใช้งานได้ 50% ของทรัพยากรที่มีอยู่ โดยการทดลองนี้แบ่งออกเป็น 4 รูปแบบได้แก่

1. เพิ่ม Database อินสแตนท์ เมื่อ CPU usage > 70%
2. เพิ่ม Database อินสแตนท์ เมื่อ CPU usage > Critical Point
3. เพิ่ม Database อินสแตนท์ เมื่อ CPU usage > Critical Point - 5%
4. เพิ่ม Database อินสแตนท์ เมื่อ CPU usage > Critical Point - 5% และ Disk usage > 75%

จาก ข้อมูลข้างต้น Critical Point ในการทดลองนี้คือ 67% ซึ่งได้มาจากการหาจุดตัดของกราฟระหว่าง Response time/จำนวน Request ใน 1 วินาทีและงานวิจัยนี้ได้กำหนด QoS ไว้ที่ 150 Millisecond จากการทดลองพบว่าการจำลองการขยายตัวของ อินสแตนท์ ในแบบที่ 3 มี Latency ที่มากกว่า 150 Millisecond เพียง 30% ซึ่งสามารถสรุปได้ว่าการจำลองแบบที่ 3 นั้นให้ผลดีที่สุดเนื่องจากเกิดการขยายตัวของ อินสแตนท์ เร็วกว่าแบบจำลองอื่น (Delnat, Truyen, Rafique, Van Landuyt, & Joosen, 2018)

(4) บทความวิจัยเรื่อง A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resourcesงานวิจัยนี้ได้นำ Cluster scheduler ของบริษัทต่างๆ มาเปรียบเทียบค่าใช้จ่ายๆ โดย Cluster scheduler มีดังต่อไปนี้ Kubernetes, COCA, Stratus และ HTAS โดยการจำลองการทดสอบแบบ Batch processing เพื่อเปรียบเทียบหาค่าใช้จ่าย ซึ่งแบ่งการเปรียบเทียบเป็น 4 แบบได้แก่ Stable, Growing, Cycle และ On/Off จากผลการทดลองพบว่า HITAS ใช้ค่าใช้จ่ายน้อยที่สุดและมีการใช้ทรัพยากรน้อยที่สุดเมื่อเปรียบเทียบกันทั้ง 4 แบบ (Chung, Park, & Ganger, 2018)

(5) บทความวิจัยเรื่อง Workload Characterization for Microservices งานวิจัยนี้ได้นำเสนอการเปรียบประสิทธิภาพของสถาปัตยกรรมแบบ Monolith และ Microservices โดยทำการเปรียบเทียบประสิทธิภาพการทำงานระหว่าง Node.js และ Java โดยการจำลองแอปพลิเคชันที่ชื่อว่า Acme Air ซึ่งการเปรียบเทียบแบ่งออกเป็นการติดตั้งซอฟต์แวร์แบบ Bare mental

Docker-Host Configuration และ Docker-Bridge Configuration ซึ่งการทดลองนี้ใช้ Apache JMeter ในการสร้างโหลดเข้าสู่แอปพลิเคชัน

โดยงานวิจัยนี้สรุปผลว่าการนำสถาปัตยกรรม Microservices มาใช้ซึ่งพัฒนาโดยภาษา Node.js และ Java นั้นส่งผลให้ Throughput ลดลงถึง 79.2% และ 70.2% ตามลำดับและยิ่งไปกว่านั้น Microservices มี Path length หรือจำนวนครั้งที่ CPU สั่งการมากกว่า Monolith 3-3.05 Times/Request (Ueda, Nakaike, & Ohara, 2016)

(6) บทความวิจัยเรื่อง Leveraging Microservices architecture by using Docker technology งานวิจัยนี้ได้อธิบายถึงแนวคิดของสถาปัตยกรรม Microservices และการนำ Docker technology มาใช้เพื่อสนับสนุนการสร้างสถาปัตยกรรมแบบ Microservices โดยงานวิจัยนี้ได้กล่าวถึงข้อดีของการนำมา Docker มาใช้กับสถาปัตยกรรมแบบ Microservices โดยมีรายละเอียดดังต่อไปนี้

1. Accelerate automation

Docker technology เหมาะแก่การทำแอปพลิเคชันที่สามารถทำงานได้อย่างอัตโนมัติ เช่น การทดสอบแอปพลิเคชันหรือการส่งมอบแอปพลิเคชัน เนื่องจากสามารถเขียนอัลกอลิทึมได้ตามความต้องการของผู้ใช้งาน

2. Accelerate the independency

Docker container มีสภาพแวดล้อมจำลองที่แยกออกจากกันชัดเจน ซึ่งทำให้ทีมพัฒนานั้นสามารถจัดการกับภาษาหรือซอฟต์แวร์ได้อย่างอิสระ

3. Accelerate portability

Docker container จัดการกับแอปพลิเคชันโดยการนำแอปพลิเคชันใส่เข้าไปใน Docker container ซึ่งมีสภาพแวดล้อมจำลองเป็น Linux ซึ่งทีมพัฒนาไม่ว่าจะเป็นผู้ทดสอบหรือนักพัฒนาสามารถนำ Container ไปใช้โดยอิสระจากกัน

4. Accelerate resource utilization

Docker technology ใช้ทรัพยากรน้อยกว่าการใช้ Virtual machine แบบปกติช่วยให้สามารถสร้าง อินสแตนท์ ได้จำนวนมาก

5. Secured

Docker เปิดให้นักพัฒนาสามารถจัดการกับความปลอดภัยของ Container ได้อย่างอิสระผ่าน Docker file และ Docker-compose (Jaramillo, Nguyen, & Smart, 2016)

(7) บทความวิจัยเรื่อง Deploying Microservices based Applications with Kubernetes: Experiments and Lessons learned งานวิจัยนี้ได้กล่าวถึงการนำแนวคิดของสถาปัตยกรรมแบบ Microservices มาใช้ร่วมกับ Kubernetes เพื่อเพิ่ม High availability ให้กับแอปพลิเคชัน โดยงานวิจัยนี้ได้ทำการทดสอบเหตุกการณ์จำลองว่า Kubernetes สามารถใช้เวลาเท่าใดในการ Self-healing เพื่อให้ Pods และ Nodes กลับมา Ready จากสถานะ Failure โดยการทดลองนี้แบ่งออกเป็น 2 เหตุการณ์ คือ การจดลองให้ Pods เกิดสถานะ Failure และ Node เกิดสถานะ Failure ซึ่งเวลาที่ใช้ในการวัดแบ่งออกเป็น 4 ประเภทดังต่อไปนี้

1. Reaction time เวลาที่ Kubernetes สามารถเริ่มตรวจจับได้ว่า Pods หรือ Node เกิดการทำงานไมปกติ

2. Repair time เวลาตั้งแต่ Kubernetes เริ่มตรวจจับสถานะผิดปกติจนถึงเวลาที่ Nodes หรือ Pods กำลังถูกซ่อมแซม

3. Recovery time เวลาตั้งแต่ Kubernetes เริ่มตรวจจับสถานะผิดปกติจนถึงเวลาที่ Nodes หรือ Pods กลับมาเป็นปกติ

4. Outage time เวลารวมทั้งหมดที่ Pods หรือ Nodes ไม่อยู่ในสถานะ Ready

จากการทดลองพบว่า Kubernetes สามารถกู้คืน Pods และ Node ที่เกิดจากการจำลองเหตุการณ์ที่ทำให้สถานะของ Pods และ Nodes เกิดความผิดปกติกลับมาเป็น Ready ได้ในระยะเวลาอันสั้น (Vayghan, Saied, Toeroe, & Khendek, 2018)

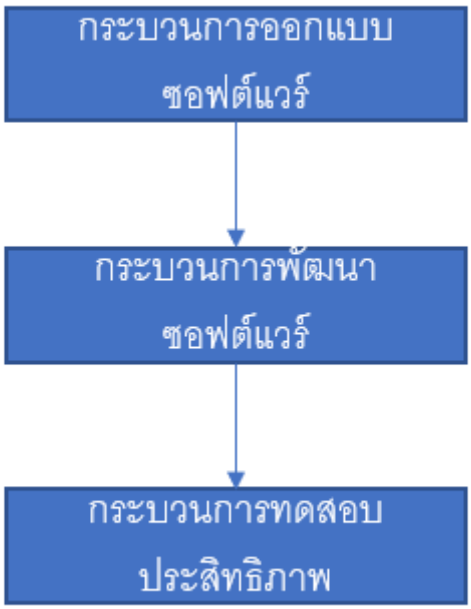
งานวิจัยนี้ประกอบไปด้วยการทบทวนวรรณกรรมทั้งหมด 6 งานวิจัยและสามารถสรุปทฤษฎีที่เกี่ยวข้องดังภาพประกอบ 7

เรื่อง	Docker	Kubernetes	Platform	Monolith	Microservices	AWS Lambda
Evaluating the Monolithic and the Microservices Architecture Pattern to Deploy Web Applications in the Cloud			Amazon web services	✔	✔	
Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservices Architectures			Amazon web services	✔	✔	✔
K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters	✔	✔	Private cloud			
A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources	✔	✔	Private cloud			
Workload Characterization for Microservices	✔	✔	IBM	✔	✔	
Leveraging Microservices architecture by using Docker technology	✔				✔	

ภาพประกอบ 7 สรุปงานวิจัยที่เกี่ยวข้อง

บทที่ 3
วิธีการดำเนินการวิจัย

จากปัญหาและทฤษฎีที่ได้กล่าวมาในบทที่ 1 และ 2 นั้นในบทนี้อธิบายถึงขั้นตอนการออกแบบและวิธีการทดสอบเพื่อหาประสิทธิภาพการทำงานของแต่โครงสร้างสถาปัตยกรรมและซอฟต์แวร์ที่นำมาใช้โดยกระบวนการต่างๆ ดังภาพประกอบ 8



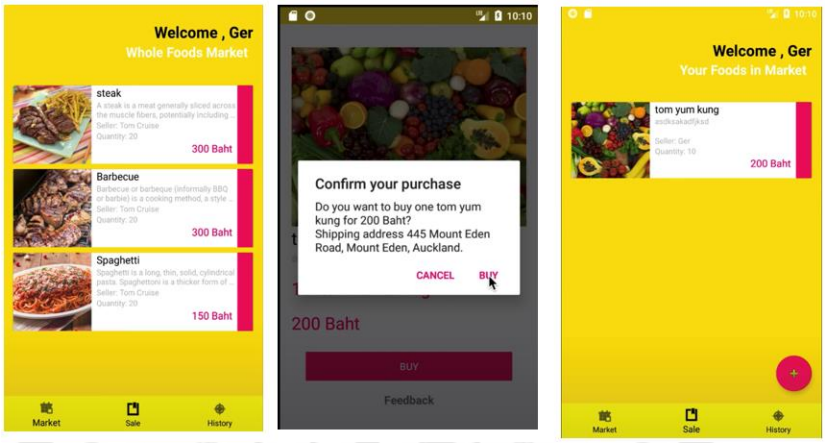
ภาพประกอบ 8 ขั้นตอนการดำเนินการวิจัย

3.1 กระบวนการออกแบบซอฟต์แวร์

ซอฟต์แวร์ที่จะใช้ในการวัดผลของการทดลองนั้นได้จำลองเหตุการณ์ของแอปพลิเคชันสะสมแต้มเพื่อแลกสินค้าโดยจำลองการเรียกดูสินค้าและการแลกสินค้าด้วยแต้มประกอบไปด้วยกระบวนการกำหนดสถาปัตยกรรม ส่วนประกอบ ส่วนประสาน และลักษณะด้านอื่นๆ ซึ่งขั้นตอนนี้ถือเป็นหัวใจหลักในการพัฒนาซอฟต์แวร์โดยมีรายละเอียดดังต่อไปนี้

3.1.1 การออกแบบส่วนต่อประสานผู้ใช้ (User interface)

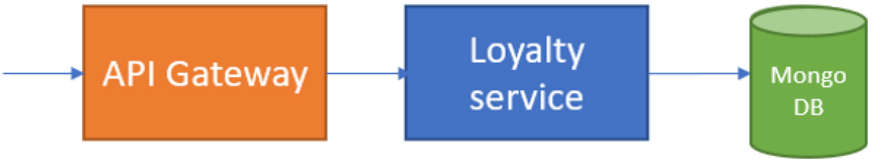
ในกระบวนการออกแบบนั้นประกอบไปด้วย User interface โดยแบ่งออกเป็น 2 สถานการณ์ได้แก่ 1) เรียกดูสินค้า และ 2) การแลกสินค้า เพื่อช่วยให้ผู้พัฒนาสามารถเข้าใจและพัฒนาได้ง่ายขึ้นโดย User interface ของแอปพลิเคชันสะสมแต้มจะมีลักษณะดังภาพประกอบ 9



ภาพประกอบ 9 ตัวอย่างหน้าจอของแอปพลิเคชันสะสมแต้มบนแอปพลิเคชันมือถือ

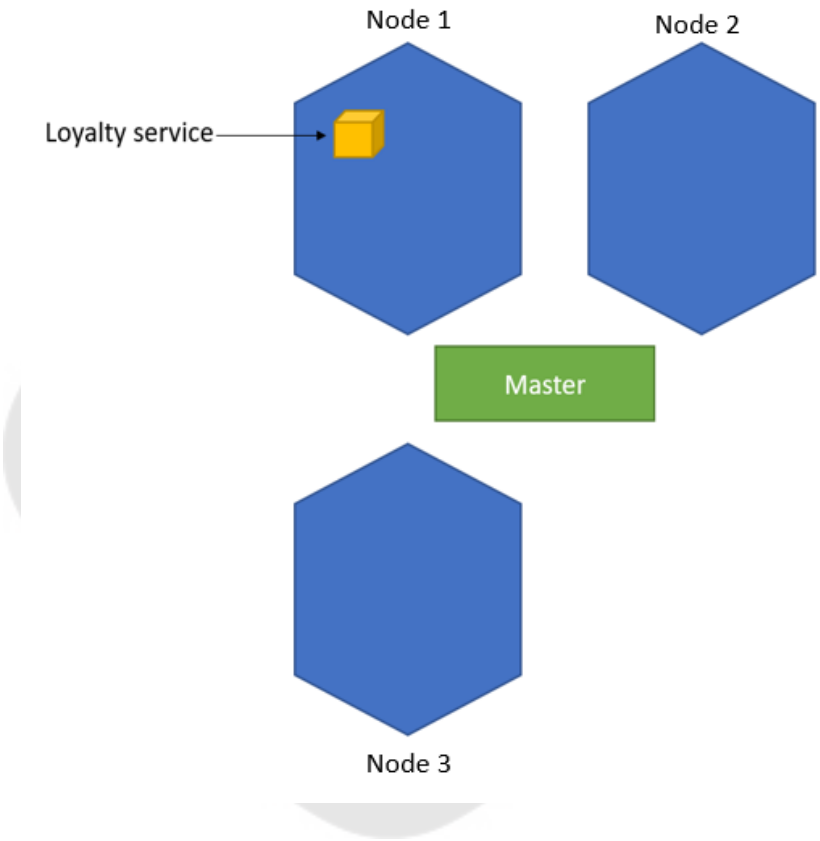
3.1.2 การออกแบบสถาปัตยกรรมแบบ Monolith

การออกแบบสถาปัตยกรรมแบบ Monolith สำหรับแอปพลิเคชันสะสมแต้มประกอบด้วย 2 ส่วนได้แก่ Loyalty service และ MongoDB ซึ่งแต่ละส่วนติดตั้งบน Docker container โดย Loyalty service นั้นมีลักษณะเป็น Single codebase โดยมีโครงสร้างสถาปัตยกรรมดังภาพประกอบ 10



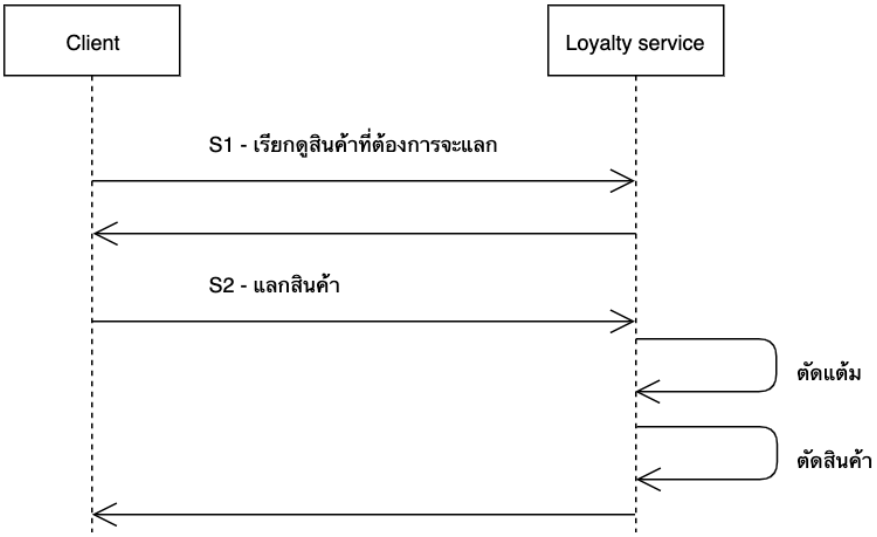
ภาพประกอบ 10 สถาปัตยกรรมแบบ Monolith โดยใช้ Docker

เมื่อออกแบบสถาปัตยกรรมแบบ Monolith เรียบร้อยแล้ว ขั้นตอนถัดไปคือการนำโครงสร้างแบบ Monolith ที่ได้ออกแบบมาข้างต้นมาพัฒนาร่วมกับ Kubernetes โดยทำการออกแบบสร้าง Cluster จำนวน 4 อินสแตนท์ ได้แก่ Master จำนวน 1 อินสแตนท์ และ Node จำนวน 3 อินสแตนท์ ซึ่งจะกำหนดให้ Pod แต่ละตัวมีขนาดของ CPU คิดเป็น 500 millicores และจะทำการ Scale out เมื่อ Node ที่มีจำนวนโหลดมากกว่า 50% CPU ดังภาพประกอบ 11



ภาพประกอบ 11 สถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

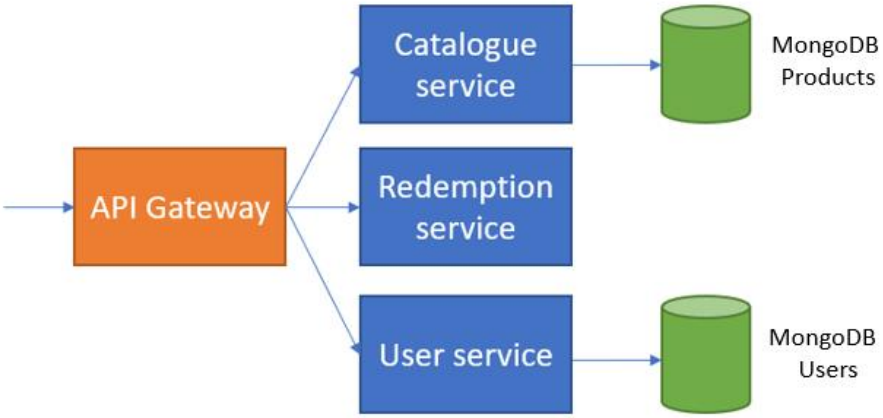
โครงสร้างสถาปัตยกรรมแบบ Monolith ของแอปพลิเคชันสะสมแต้มนั้นประกอบไปด้วยหลายฟังก์ชัน ดังนั้นแอปพลิเคชันจะมีชื่อว่า Loyalty service หรือแอปพลิเคชันสะสมแต้ม เพื่อให้ครอบคลุมทุกฟังก์ชันการทำงานซึ่งมีลำดับขั้นตอนการทำงานดังภาพประกอบ 12



ภาพประกอบ 12 ลำดับการทำงานของสถาปัตยกรรมแบบ Monolith

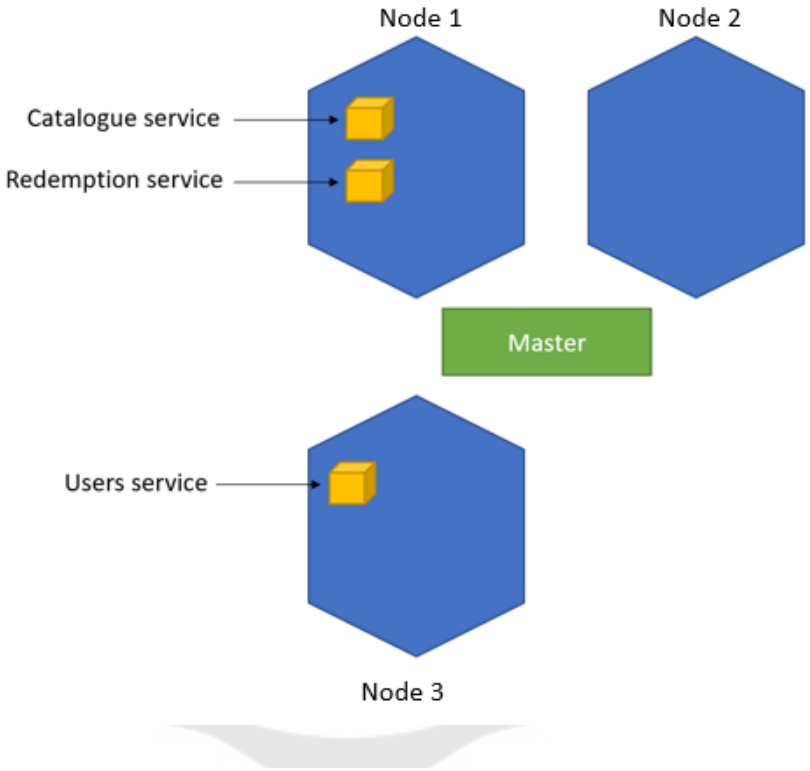
3.1.3 การออกแบบสถาปัตยกรรมแบบ microservices

แอปพลิเคชันสะสมแต้มเพื่อแลกสินค้าบนแนวคิดแบบโดยแบ่งออกเป็น 3 โดเมนซึ่งแต่ละโดเมนมีลักษณะเหมือนกับฟังก์ชันใน Monolith แต่ทำหน้าที่เฉพาะส่วนที่แต่ละโดเมนรับผิดชอบเท่านั้น ประกอบด้วยโดเมน 1) Users 2) Catalogues และ 3) Redemption โดยติดตั้งบน Docker container ที่มีโครงสร้างสถาปัตยกรรมดังภาพประกอบ 13



ภาพประกอบ 13 สถาปัตยกรรมแบบ Microservices โดยใช้ Docker

เมื่อทำการออกแบบสถาปัตยกรรมแบบ Microservices เรียบร้อยแล้ว ขั้นตอนถัดไป คือการนำโครงสร้างสถาปัตยกรรมแบบ Microservices ที่ได้ออกแบบมาข้างต้นมาพัฒนาร่วมกับ Kubernetes โดยทำการออกแบบสร้าง Cluster จำนวน 4 อินสแตนท์ ได้แก่ Master จำนวน 1 อินสแตนท์ และ Node จำนวน 3 อินสแตนท์ ซึ่งจะกำหนดให้ Pod แต่ละตัวมีขนาดของ CPU คิดเป็น 50% ของ 1 Core CPU และจะทำการ Scale out เมื่อ Node ที่มีจำนวนโหลดมากกว่า 50% CPU เช่นเดียวกับ Monolith ดังภาพประกอบที่ 14



ภาพประกอบ 14 สถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

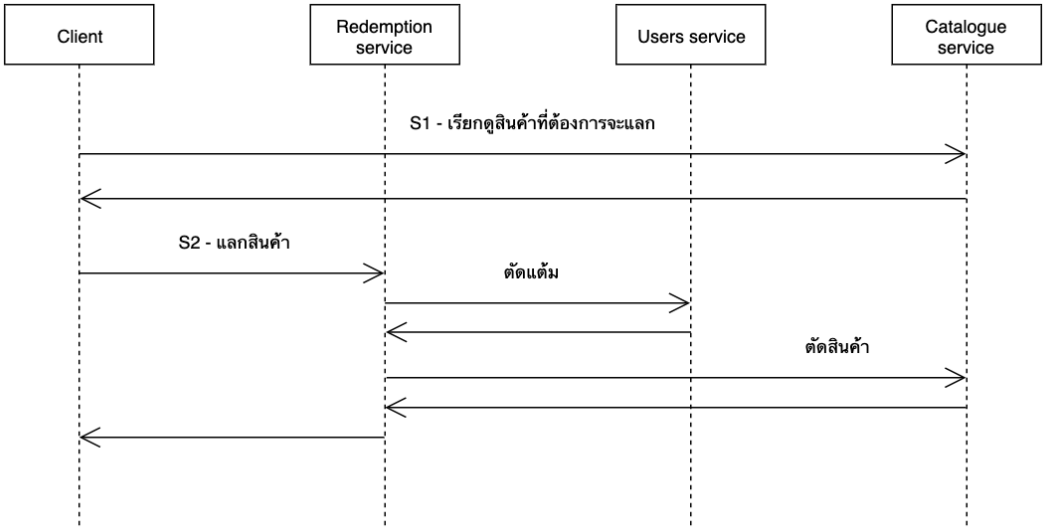
สถาปัตยกรรมแบบ Microservices นั้นประกอบไปด้วย 3 เซอร์วิส ที่ทำหน้าที่แตกต่างกันออกไปและถูกแยกออกมาเป็นซอฟต์แวร์คนละชุด ได้แก่

1) User service ทำหน้าที่จัดเก็บและอัพเดทข้อมูลของลูกค้า ซึ่งการเรียกใช้จะผ่าน API (Application program interface)

2) Catalogues service ทำหน้าที่จัดเก็บสินค้าซึ่งผู้ที่ต้องการเรียกดูสินค้าต่างๆ จะต้องเรียกมาใช้บริการผ่าน API (Application program interface)

3) Redemption service ทำหน้าที่เป็นตัวกลางในการแลกสินค้าโดยจะทำหน้าที่ติดต่อกับ Catalogues service และ User service เพื่อทำการตัดแต้มและตัดสินค้าผ่าน REST (Representational state transfer)

โครงสร้างสถาปัตยกรรมแบบ Microservices ของแอปพลิเคชันสะสมแต้มนั้นประกอบไปด้วย S1. เรียกดูสินค้าที่ต้องการจะแลกโดยแอปพลิเคชันเรียกข้อมูลสินค้าจาก Catalogues service และ S2. เมื่อลูกค้าแลกสินค้าระบบจะทำการตัดแต้มของลูกค้าที่ User services หลังจากนั้นทำการตัดสินค้าออกจากคลังดังภาพประกอบที่ 15



ภาพประกอบ 15 ลำดับการทำงานของสถาปัตยกรรมแบบ Microservices

3.1.3 การออกแบบวิธีการรับส่งข้อมูล

การออกแบบสถาปัตยกรรมโดยทั่วไปนั้นแอปพลิเคชันที่ทำหน้าที่เป็น User interface หรือส่วนที่ทำหน้าที่ติดต่อกับผู้ใช้งานจะต้องทำการรับส่งข้อมูลกับเซิร์ฟเวอร์ผ่าน REST (Representational state transfer) โดยในขั้นตอนนี้จะอธิบายถึงการออกแบบ API (Application Program Interface) ของสถาปัตยกรรมแบบ Monolith และ Microservices

3.1.3.1 Monolith API

จากภาพประกอบ 11 แสดงให้เห็นถึงการรับส่งข้อมูลระหว่าง Client กับ Loyalty service ซึ่งประกอบไปด้วย API (Application Program Interface) จำนวน 2 เซอร์วิส ได้แก่

@GET /v1/loyalty/catalogues ใช้สำหรับเรียกข้อมูลสินค้าจากเซิร์ฟเวอร์

@POST /v1/loyalty/redeem ใช้สำหรับแลกสินค้า

3.1.3.2 Microservices API

จากภาพประกอบ 14 แสดงให้เห็นถึงการรับส่งข้อมูลระหว่าง Client ที่ติดต่อเซอร์วิสต่างๆ โดยประกอบไปด้วย API (Application Program Interface) ที่รับส่งข้อมูลจาก Client และ API สำหรับรับส่งข้อมูลระหว่างเซอร์วิสด้วยกันเองซึ่งประกอบไปด้วย API (Application Program Interface) จำนวน 4 เซอร์วิส ได้แก่

@GET /v1/loyalty/catalogues ซึ่งอยู่ในโดเมนของ Catalogue service ใช้เพื่อทำการเรียกข้อมูลสินค้าจากเซิร์ฟเวอร์

@POST /v1/loyalty/deductItem ซึ่งอยู่ในโดเมนของ Catalogue service ใช้เพื่อทำการหักสินค้าที่ถูกแลกออกจากคลัง

@POST /v1/loyalty/deductPoint ซึ่งอยู่ในโดเมนของ Users service ใช้เพื่อทำการหักแต้มหลังจากทำการแลกสินค้า

@POST /v1/loyalty/redeem ซึ่งอยู่ในโดเมนของ Redemption service ใช้เพื่อทำการแลกสินค้า

3.1.4 **กระบวนการออกแบบ Cluster**

การออกแบบและการสร้าง อินสแตนท์ บน Amazon webservices ซึ่งในงานวิจัยนี้ประกอบไปด้วย EC2 ทั้งหมดจำนวน 7 อินสแตนท์ ได้แก่

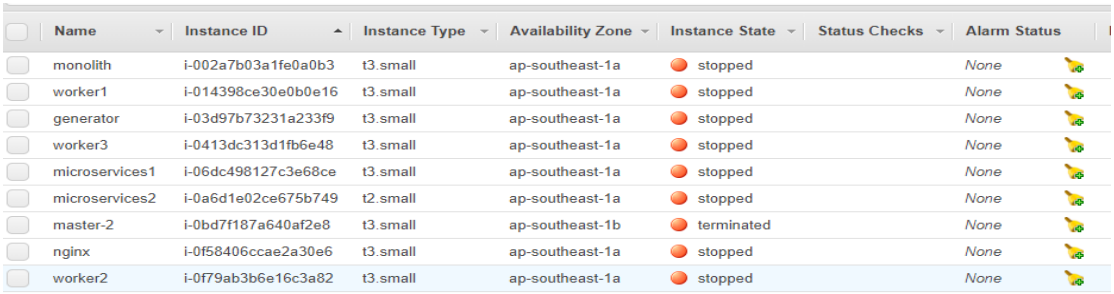
Master node จำนวน 1 อินสแตนท์ โดยทำหน้าที่ควบคุม Cluster

Node node จำนวน 3 อินสแตนท์ โดยทำหน้าที่เป็นตัวประมวลผล

API Gateway จำนวน 2 อินสแตนท์ โดยทำหน้ากระจาย Request ไปยังส่วนต่างๆ

Load test generator จำนวน 1 อินสแตนท์ โดยทำหน้าที่สร้าง Workload ให้แก่แอปพลิเคชัน

การออกแบบโครงสร้างของ Network ภายใน Cluster โดยให้ทั้งหมดอยู่ใน Private subnet โดยมีเพียง Load test generator อยู่ใน Public subnet เท่านั้น โดยรายละเอียดของประเภท อินสแตนท์ ที่ถูกใช้ในงานวิจัยนี้ถูกแสดงดังภาพประกอบ 16



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
monolith	i-002a7b03a1fe0a0b3	t3.small	ap-southeast-1a	stopped		None
worker1	i-014398ce30e0b0e16	t3.small	ap-southeast-1a	stopped		None
generator	i-03d97b73231a233f9	t3.small	ap-southeast-1a	stopped		None
worker3	i-0413dc313d1fb6e48	t3.small	ap-southeast-1a	stopped		None
microservices1	i-06dc498127c3e68ce	t3.small	ap-southeast-1a	stopped		None
microservices2	i-0a6d1e02ce675b749	t2.small	ap-southeast-1a	stopped		None
master-2	i-0bd7f187a640af2e8	t3.small	ap-southeast-1b	terminated		None
nginx	i-0f58406ccae2a30e6	t3.small	ap-southeast-1a	stopped		None
worker2	i-0f79ab3b6e16c3a82	t3.small	ap-southeast-1a	stopped		None

ภาพประกอบ 16 รายการ อินสแตนท์ ประเภทต่างๆ ที่ถูกใช้ในงานวิจัยบน Amazon webservices

3.1.5 **เครื่องมือที่ใช้ในการวิจัย**

การจำลองแอปพลิเคชันสะสมแต้มในงานวิจัยนี้ประกอบไปด้วยเครื่องมือที่ใช้ในการพัฒนาซอฟต์แวร์โดยมีรายละเอียดดังตาราง 2

ตาราง 2 เครื่องมือที่ใช้ในการพัฒนา

เครื่องมือที่ใช้ในการพัฒนา	เวอร์ชั่น	หน้าที่
Node.js	8.6.0	ภาษาที่ใช้พัฒนาแอปพลิเคชัน
MongoDB	3.4.0	ฐานข้อมูลสำหรับแอปพลิเคชัน
Docker	19.0.3	Container สำหรับติดตั้งแอปพลิเคชัน
Kubernetes	1.15.0	เครื่องมือช่วยจัดการกับ Container
Apache JMeter	5.2.1	เครื่องมือในการจำลองปริมาณการใช้งาน
Grafana	6.7.3	แดชบอร์ดสำหรับตรวจสอบการใช้งานทรัพยากร
Prometheus	2.17.2	ฐานข้อมูลที่ใช้จัดเก็บข้อมูลทรัพยากร

การจำลองแอปพลิเคชันสะสมแต้มแบบ Monolith ได้จัดเตรียม Amazon web services โดยแต่ละอินสแตนท์มีคุณสมบัติดังตาราง 3

ตาราง 3 คุณสมบัติของ AWS อินสแตนท์ ที่ใช้ในการทดสอบแบบ Monolith

หน้าที่	AWS อินสแตนท์	คุณสมบัติ	จำนวน (อินสแตนท์)
Loyalty service	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
API gateway	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
Test generator	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1

การจำลองแอปพลิเคชันสะสมแต้มแบบ Microservices ได้จัดเตรียม Amazon web services โดยแต่ละอินสแตนท์มีคุณสมบัติดังตาราง 4

ตาราง 4 คุณสมบัติของ AWS อินสแตนท์ ที่ใช้ในการทดสอบแบบ Microservices

หน้าที่	AWS อินสแตนท์	คุณสมบัติ	จำนวน (อินสแตนท์)
Users service	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
Catalogues service	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
Redemption service	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
API gateway	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
Test generator	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1

การจำลองแอปพลิเคชันสะสมแต้มแบบ Cluster ได้จัดเตรียม Amazon web services โดยแต่ละอินสแตนท์มีคุณสมบัติดังตาราง 5

ตาราง 5 คุณสมบัติของ AWS อินสแตนท์ ที่ใช้ในการทดสอบแบบ Cluster

หน้าที่	AWS อินสแตนท์	คุณสมบัติ	จำนวน (อินสแตนท์)
Master	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1
Nodes	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	3
Test generator	T3.small	2 vCPUs, 2.5 GHz, 2 GiB memory	1

3.2 กระบวนการพัฒนาซอฟต์แวร์

หลังจากการออกแบบสถาปัตยกรรมแบบ Monolith และ Microservices เรียบร้อยแล้ว ขั้นตอนถัดไปคือกระบวนการพัฒนาซอฟต์แวร์ โดยกระบวนการพัฒนาซอฟต์แวร์ในงานวิจัยนี้ ประกอบไปด้วย 4 ส่วน ได้แก่ พัฒนา (Develop), สร้าง (Build), ส่งมอบ (Ship) และ ใช้งาน (Deploy)

3.2.1 การพัฒนา

ในส่วนนี้เป็นการพัฒนาแอปพลิเคชันฝั่ง Backend ที่แบ่งออกเป็น 2 สถาปัตยกรรม ได้แก่ Monolith และ Microservices โดยซอฟต์แวร์ทั้งหมดจะถูกพัฒนาด้วย Node.js ดังภาพประกอบ 17 ซึ่งประกอบไปด้วยซอฟต์แวร์ทั้งหมด 4 ชุดจำแนกตามสถาปัตยกรรม

สถาปัตยกรรมแบบ Monolith

1) Loyalty service (Monolith)

สถาปัตยกรรมแบบ Microservices

1) Users service (Microservices)

2) Catalogue service (Microservices)

3) Redemptions service (Microservices)

```
getCatalogues(){
    return new Promise<any>((resolve, reject) => {
        if (this.error) {
            return reject(this.error);
        }

        this.cataloguesModel.find({})
        .then((result) => {
            if (!result){
                return resolve();
            }
            return resolve(result);
        }).catch((error) => {
            reject(error);
        });
    });
}
redeem(customerID:string){
    return new Promise<any>((resolve, reject) => {
        if (this.error) {
            return reject(this.error);
        }

        this.usersModel.
        update({ customer_id: customerID}, { $inc: { points: -1 }})
        .then((result) => {
            if (!result){
                return resolve();
            }
            return resolve(result);
        }).catch((error) => {
            reject(error);
        });
    });
}
```

ภาพประกอบ 17 ตัวอย่างโค้ดที่ใช้ในการพัฒนาแอปพลิเคชัน

ขั้นตอนต่อไปคือการเตรียมข้อมูลใน Database โดยที่ Database จะแบ่งออกเป็น 2 ส่วน ได้แก่ Database สำหรับสถาปัตยกรรมแบบ Monolith และ Database สำหรับสถาปัตยกรรมแบบ Microservices โดยมีการแบ่ง Collection ดังต่อไปนี้

1) Users collection จัดเก็บข้อมูลลูกค้าจำนวน 1,000 Documents

2) Products collection จัดเก็บข้อมูลสินค้าจำนวน 3 Documents

3.2.2 การสร้างและส่งมอบ

การพัฒนาซอฟต์แวร์โดยใช้ Docker นั้นซอฟต์แวร์ที่เสร็จแล้วจะถูกนำไปบรรจุใส่ Docker image ผ่าน Dockerfile เพื่อทำการส่งมอบ Image ไปยัง Docker hub ซึ่งทำหน้าที่เป็นศูนย์กลางในการจัดเก็บและเผยแพร่ Image โดยการส่งมอบ Image นั้นสามารถอธิบายได้ดังภาพประกอบ 18

ภาพประกอบ 18 ขั้นตอนการส่องมอบ Docker image ไปยัง Amazon webservices

Dockerfile คือ text document ที่ใช้ในการเขียนคำสั่งเพื่อสร้าง Docker image โดยแสดงตัวอย่างดังภาพประกอบ 19

```
FROM node-base:1.0.1
LABEL maintainer="NAPAWIT"

RUN mkdir -p /app
WORKDIR /app

COPY . /app

EXPOSE 3000
CMD ["npm", "run", "js-start"]
```

ภาพประกอบ 19 ตัวอย่าง Dockerfile

3.2.3 การนำไปใช้งาน

ขั้นตอนของการนำไปใช้งานหลังจากส่งมอบ Docker image ไปยัง Docker hub แล้ววิธีการ Deploy ในส่วนของ Docker นั้นใช้ docker-compose.yml ซึ่งเป็น Configuration โดยทำการตั้งค่าเกี่ยวกับเวอร์ชันของซอฟต์แวร์ที่ใช้และการติดต่อกันระหว่างเครือข่ายโดยผู้ใช้จะต้องเป็นผู้กำหนด สำหรับติดตั้ง Docker image ดังตัวอย่างภาพประกอบ 20

```yaml
version: "3"

services:
  loyalty-mongo:
    container_name: loyalty-mongo
    image: mongo:3.4.10
    ports:
    - "27017:27017"
    volumes:
    - /Users/ger/loyalty-monolith/db:/data/db
    logging:
      driver: "json-file"
      options:
        max-size: "8k"
  monolith-loyalty-service:
    container_name: monolith-loyalty-service
    image: monolith-loyalty:1.0.0
    ports:
    - "9000:3000"
    expose:
    - 9000
    volumes:
    - /Users/ger/loyalty-monolith/loyalty-service/app/config:/app/config
    - /Users/ger/loyalty-monolith/loyalty-service/app/logs:/app/logs
    logging:
      driver: "json-file"
      options:
        max-size: "8k"
    environment:
    - NODE_CONFIG_DIR=/app/config
    - NODE_ENV=development
```

ภาพประกอบ 20 ตัวอย่าง docker-compose.yml

ไฟล์ docker-compose.yml จะเป็นการระบุว่า Docker image ที่จะเรียกมาใช้งานจาก Docker hub นั้นมีชื่อว่าอะไรและจะทำการติดตั้งด้วย Port ใด เป็นต้น

วิธีการ Deploy โดยใช้ Kubernetes โดยการ Deploy ซอฟต์แวร์โดยใช้ Kubernetes นั้นจะไปประกอบไปด้วย 2 ส่วน ได้แก่ Pod และ Services ดังภาพประกอบตัวอย่างที่ 21

```yaml
apiVersion: v1
kind: Service
metadata:
  name: user-mongo-microservice
  labels:
    name: user-mongo-microservice
    owner: Napawit
    version: "1.0"
    module: WebServer
    environment: development
spec:
  selector:
    name: user-mongo-microservice
    owner: Napawit
    version: "1.0"
    module: WebServer
    environment: development
  type: NodePort
  ports:
   - nodePort: 30166
     port: 27017
     name: http
     targetPort: 27017
     protocol: TCP
---
apiVersion: "v1"
kind: Pod
metadata:
  name: user-mongo-microservice
  labels:
    name: user-mongo-microservice
    owner: Napawit
    version: "1.0"
    module: WebServer
    environment: development
spec:
  containers:
    - name: user-mongo-microservice
      image: mongo:3.4.10
      ports:
        - containerPort: 27017
          protocol: TCP
```

ภาพประกอบ 21 ตัวอย่าง example-deployment.yml

การติดตั้ง Pod และ service ของ Kubernetes งานวิจัยนี้ได้ใช้คุณสมบัติของ Kubernetes คือ Horizontal pod autoscaling ซึ่งติดตั้งให้กับทุก Pod ที่อยู่ใน Cluster โดยมีการกำหนดกฎของการ Scale out เป็นดังนี้

1) Pod มีขนาดเป็น 500 millicores ของ CPU ยกตัวอย่างเช่น 500 millicores ของ 1 Core CPU จะเท่ากับ 50% ของ CPU ดังภาพประกอบ 22

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loyalty-catalogues-service
  labels:
    name: loyalty-catalogues-service
    owner: Napawit
    version: "1.0"
    module: WebServer
    environment: development
spec:
  replicas: 1
  selector:
    matchLabels:
      name: loyalty-catalogues-service
      owner: Napawit
      version: "1.0"
      module: WebServer
      environment: development
  template:
    metadata:
      labels:
        name: loyalty-catalogues-service
        owner: Napawit
        version: "1.0"
        module: WebServer
        environment: development
    spec:
      containers:
        - name: loyalty-catalogues-service
          image: napawitto/microservice-loyalty-catalogue:1.0.4
          resources:
            requests:
              cpu: "500m"
          ports:
          - containerPort: 3000
            protocol: TCP
```

ภาพประกอบ 22 ตัวอย่าง example-deployment.yml ที่มีการกำหนดขนาดของ CPU

2) Pod จะ Scale out ก็ต่อเมื่อมีการใช้ CPU มากกว่า 50% โดยกำหนดให้ Pod เริ่มต้นมีขนาดเท่ากับ 1 ตัวและสามารถ scale out ได้มากที่สุด 10 ตัว ดังตัวอย่างคำสั่งต่อไปนี้

- kubectl autoscale deployment/app-monolith --min=1 —max=10 —cpu-percent=50

3.3 วิธีการทดลอง

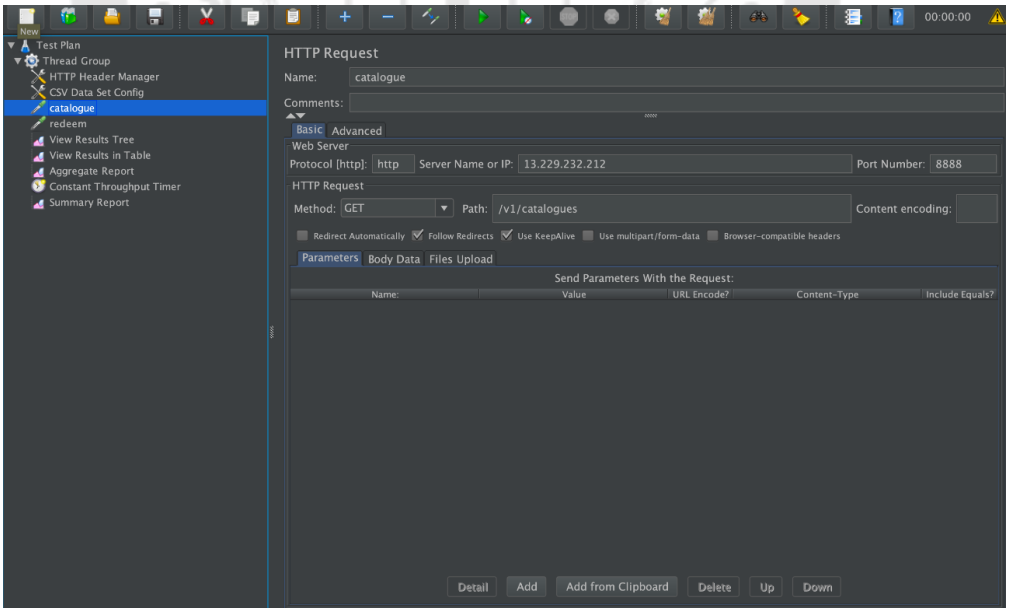
การเปรียบประสิทธิภาพของสถาปัตยกรรมแบบ Monolith และ Microservices โดยโหลดผ่าน REST (Representational state transfer) ไปยังแอปพลิเคชันจำนวน 300 Transactions/second โดยมีการเพิ่ม 1 Thread ทุกๆ 10 วินาที จนถึงจำนวน 30 Threads โดยการทดสอบใช้ระยะเวลา 10 นาที ทดสอบโดยใช้ Apache JMeter ซึ่งแบ่งเป็นการเรียกดูสินค้าและการทำรายการแลกสินค้า โดยวิธีการวัดผลแบ่งออกเป็น 2 ส่วนดังนี้

1. เปรียบเทียบ Response time ระหว่าง Monolith และ Microservices โดยใช้ Docker บน Amazon webservice โดยใช้ EC2 ที่มี Ubuntu OS 1.8.4 T3.Small 2 vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory, EBS

2. เปรียบเทียบ Response time ระหว่าง Monolith และ Microservices โดยใช้ Kubernetes บน Amazon webservice ซึ่งทำการติดตั้ง Kubernetes cluster ไว้โดยมีการทำ Horizontal pods autoscaler เมื่อใช้ CPU เกิน 50% ประกอบไปด้วย EC2 จำนวน 4 อินสแตนท์ ซึ่งมีคุณสมบัติดังต่อไปนี้ Master node ที่มีคุณสมบัติ Ubuntu OS 1.8.4 T3.Small 2 vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory, EBS และ 3 Node nodes โดยมีคุณสมบัติเดียวกันกับ Master node

3.4 วิธีการวิเคราะห์ผล

การทดสอบใช้ Apache JMeter ในการทดสอบโหลดขนาด 300 Transactions/second ระยะเวลา 10 นาที จำนวน 3 ครั้ง โดยตัวแปรที่ใช้วัดประสิทธิภาพการใช้งานคือ Throughput และ Average response time ตัวอย่างการตั้งค่าของ Apache JMeter แสดงดังภาพประกอบ 23



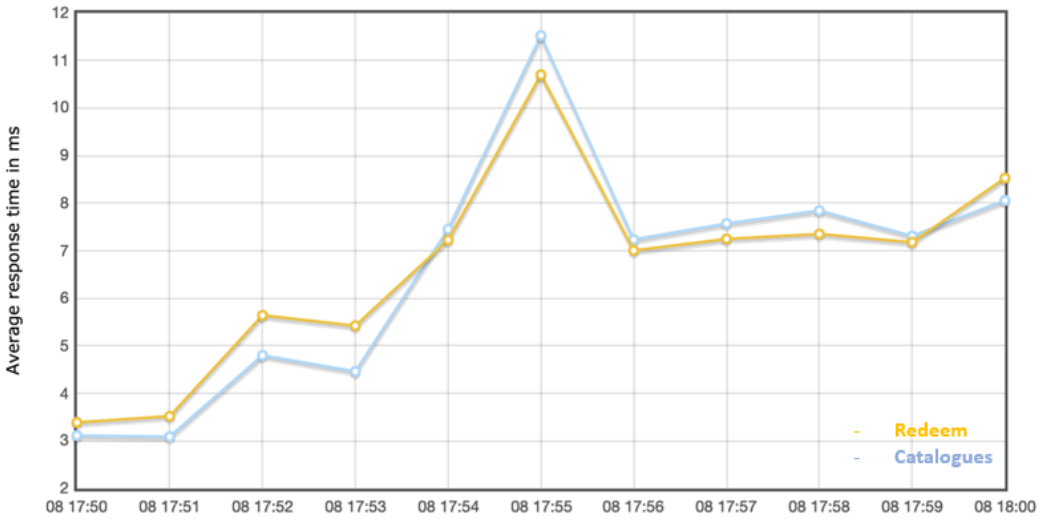
ภาพประกอบ 23 หน้าจอการตั้งค่า Apache Jmeter

บทที่ 4
ผลการศึกษา

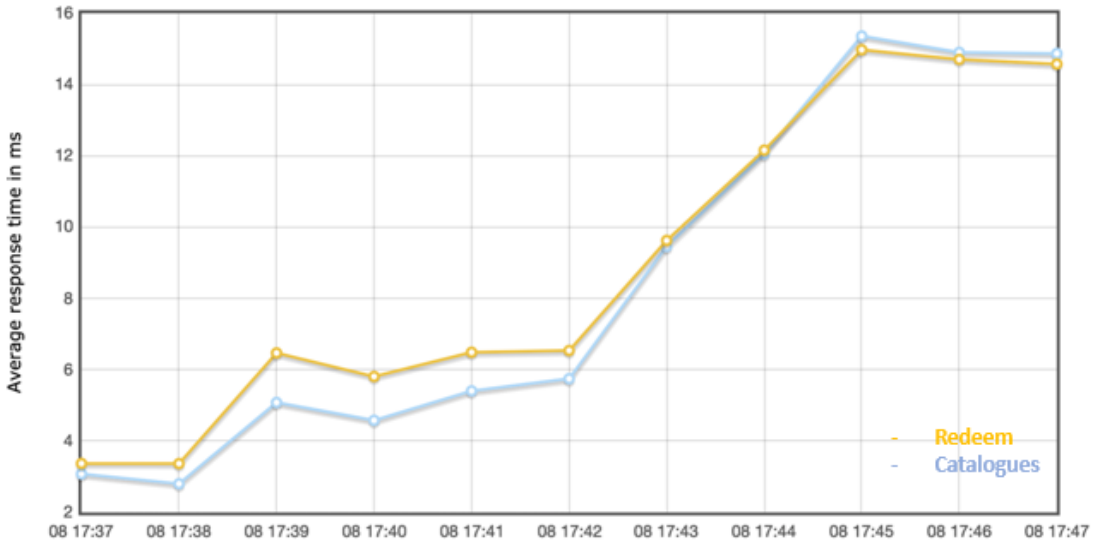
จากการทดลองการเปรียบเทียบประสิทธิภาพด้วยสถาปัตยกรรมแบบ Monolith และ Microservices โดยใช้ Docker และ Kubernetes เห็นได้ชัดว่าการนำสถาปัตยกรรมและเทคโนโลยีที่นำมาประยุกต์ใช้ร่วมกันนั้นสามารถเพิ่มความสามารถให้แอปพลิเคชันที่จำลองขึ้นมาทำงานได้ดียิ่งขึ้นและไม่ได้ส่งผลกับประสิทธิภาพของแอปพลิเคชันเดิม

4.1 ผลการทดสอบของสถาปัตยกรรมแบบ Monolith โดยใช้ Docker

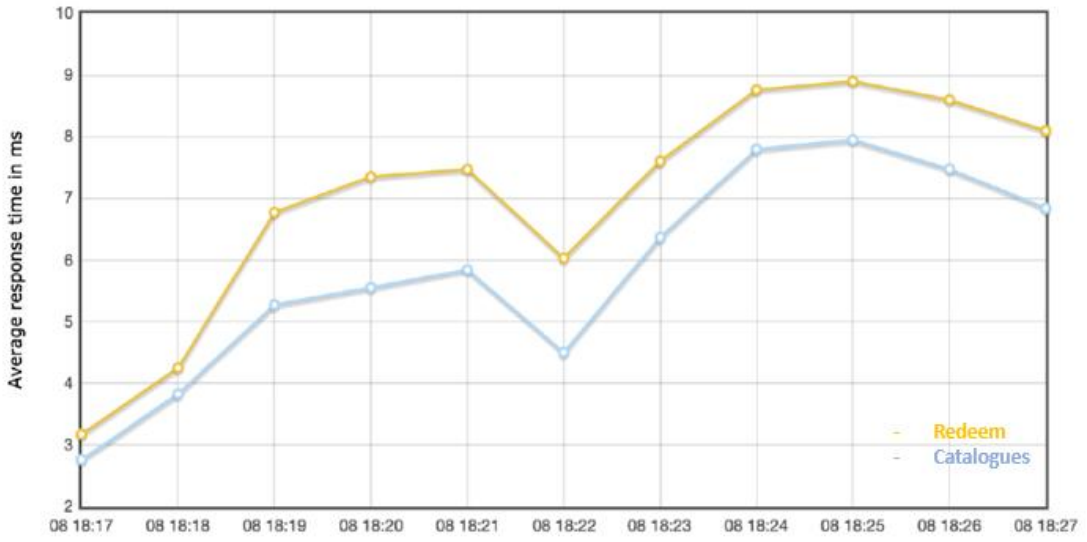
การทดสอบ Monolith โดยใช้ Docker จำนวน 3 ครั้ง พบว่า Response time และ Throughput ของการทดสอบมีค่าเพิ่มสูงขึ้นเรื่อยๆ เนื่องจากจำนวน Thread มีการเพิ่มขึ้นทุกๆ 10 วินาที และจะมีจำนวนครบ 30 Threads ตามที่กำหนดในนาทีที่ 5 โดยผลการทดสอบทั้ง 3 ครั้ง แสดงดังภาพประกอบ 24-26



ภาพประกอบ 24 ผลการทดสอบครั้งที่ 1 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Docker



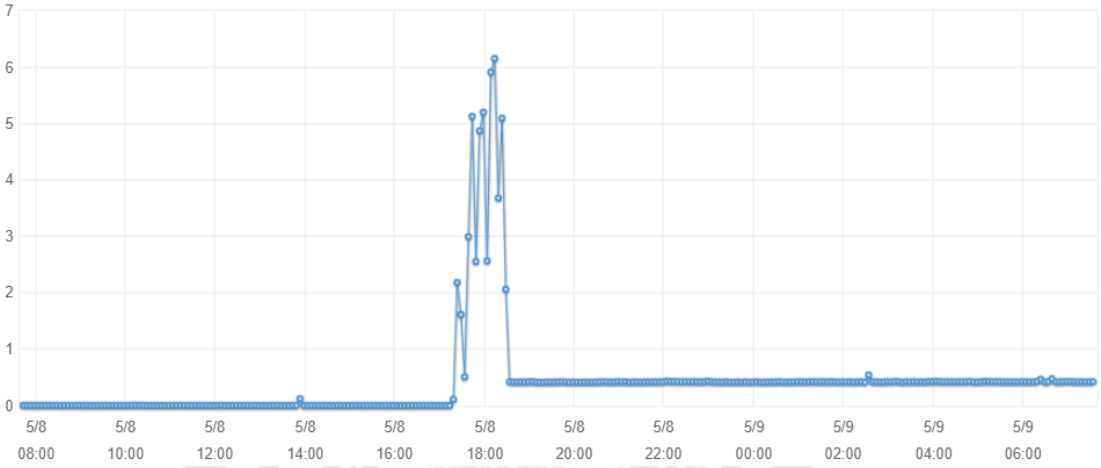
ภาพประกอบ 25 ผลการทดสอบครั้งที่ 2 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Docker



ภาพประกอบ 26 ผลการทดสอบครั้งที่ 3 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Docker

จากภาพประกอบข้างต้นจะสังเกตุเห็นได้ว่านาทีที่ 5 ของการทดสอบทั้ง 3 ครั้งจะเกิด Response time ที่เปลี่ยนไปในแต่ละการทดลอง ซึ่งปัจจัยที่ส่งผลให้มีการเปลี่ยนแปลงนั้นพบว่าการทำงานของ Amazon webservices ของ T3.small ซึ่งเป็น อินสแตนท์ type ประเภท Burstable Performance อินสแตนท์ จะช่วยเพิ่มประสิทธิภาพให้กับแอปพลิเคชันให้กับ CPU โดย

การใช้คะแนน CPU credit usage โดยการทดสอบเกิดขึ้นในช่วงเวลา 17.37 – 18.27 นาฬิกา โดยแสดงการใช้งาน CPU credit usage ดังภาพประกอบ 27



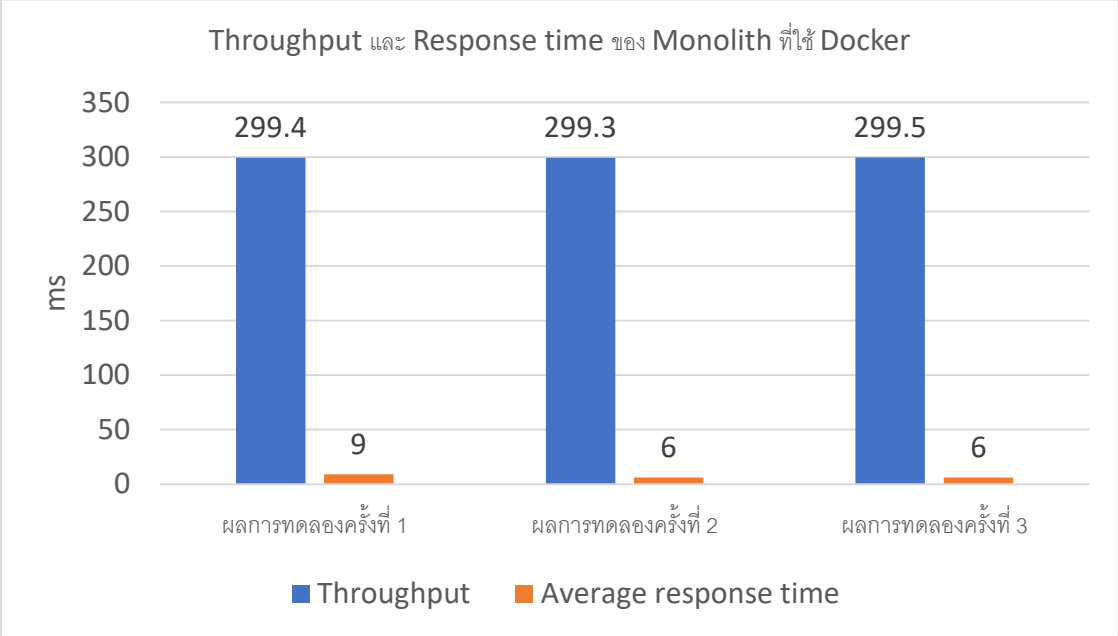
ภาพประกอบ 27 การใช้งาน CPU credit usage ในขณะทดสอบของสถาปัตยกรรม Monolith โดยใช้ Docker

จากการจำลองทดสอบแอปพลิเคชันสะสมแต้มที่พัฒนาด้วยสถาปัตยกรรมแบบ Monolith ด้วย Docker บน Amazon webservices โดยทำการทดสอบโหลด 300 Transactions/second ซึ่งมี Response time เฉลี่ยอยู่ที่ 7 Millisecond และ Throughput 299.40 Transactions/second โดยผลการทดลองทั้ง 3 ครั้งแสดงดังตาราง 6

ตาราง 6 ผลการทดลองของสถาปัตยกรรมแบบ Monolith โดยใช้ Docker

Experimental	Error %	Throughput (tps)	Response time (ms)
ผลการทดลองครั้งที่ 1	0	299.4	9
ผลการทดลองครั้งที่ 2	0	299.3	6
ผลการทดลองครั้งที่ 3	0	299.5	6
เฉลี่ย	0	299.40	7

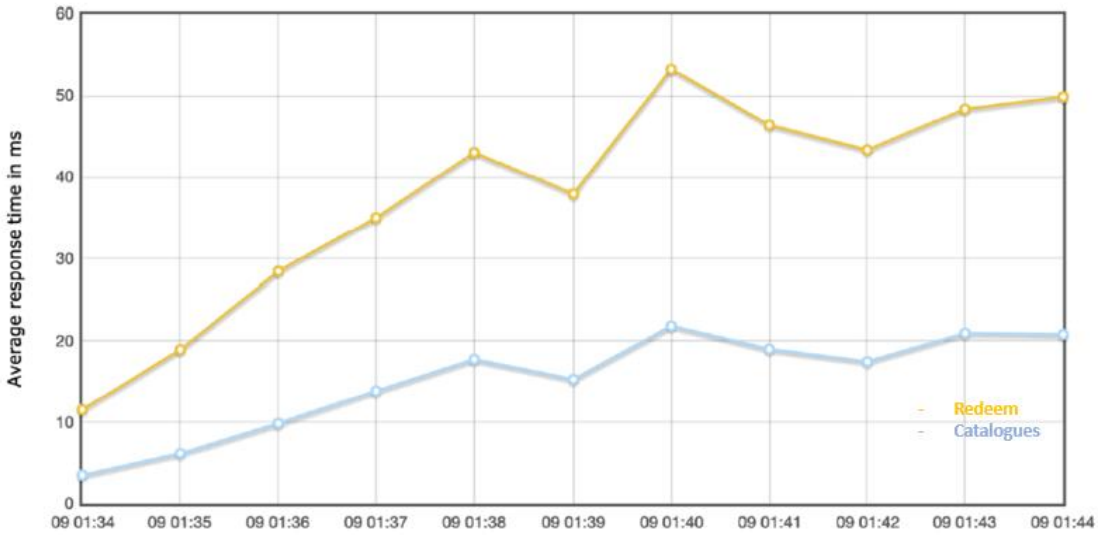
จากตาราง 6 พบว่า Throughput ของการทดลองแต่ละครั้งมีค่าที่เข้าใกล้ 300 Transactions/second ซึ่งหมายถึงระบบที่ออกแบบยังคงรองรับการปริมาณการใช้งานที่ 300 Transactions/second ตามที่กำหนดไว้และผลการเปรียบเทียบของการทดสอบแสดงดังภาพประกอบ 26



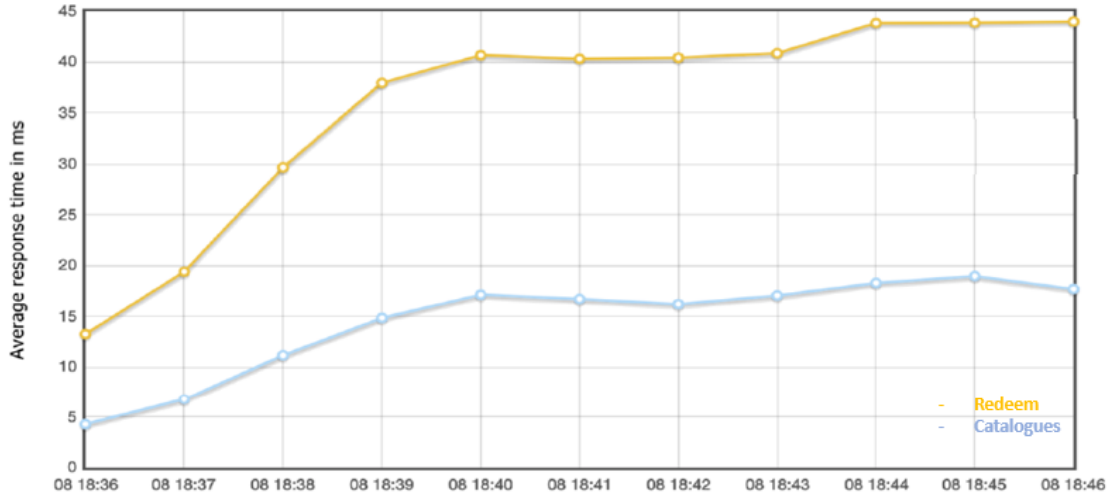
ภาพประกอบ 28 Throughput และ Response time ของ Monolith ที่ใช้ Docker

4.2 ผลการทดสอบของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker

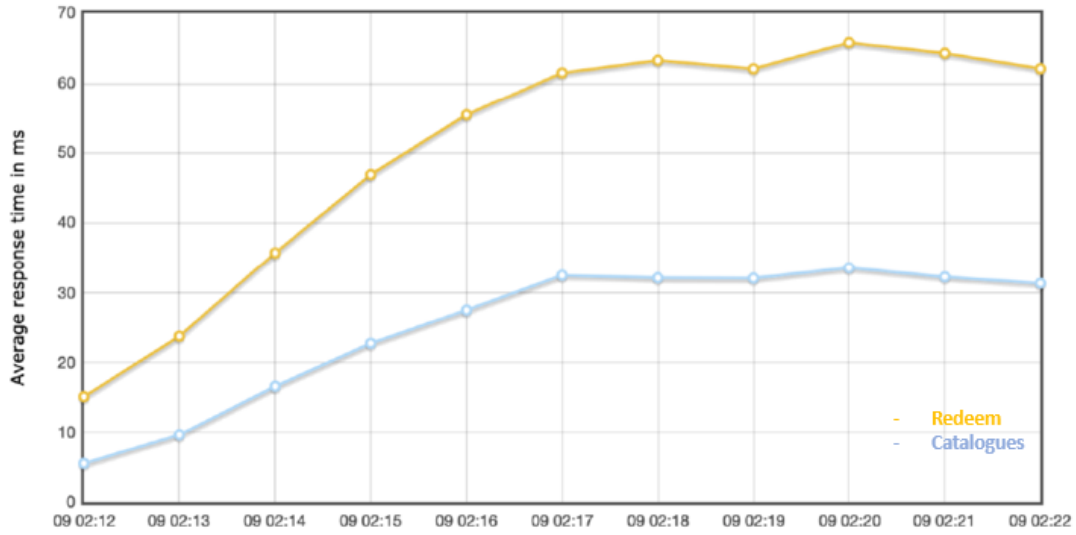
ผลการทดลองของ Microservices โดยใช้ Docker นั้นพบว่าพฤติกรรมการเพิ่ม Response time มีลักษณะคล้ายกับสถาปัตยกรรมแบบ Microservices ซึ่งเป็นผลมาจากการเพิ่มจำนวน Thread ทุกๆ 10 วินาที โดยผลการทดสอบทั้ง 3 ครั้งแสดงดังภาพประกอบ 29-31



ภาพประกอบ 29 ผลการทดสอบครั้งที่ 1 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker

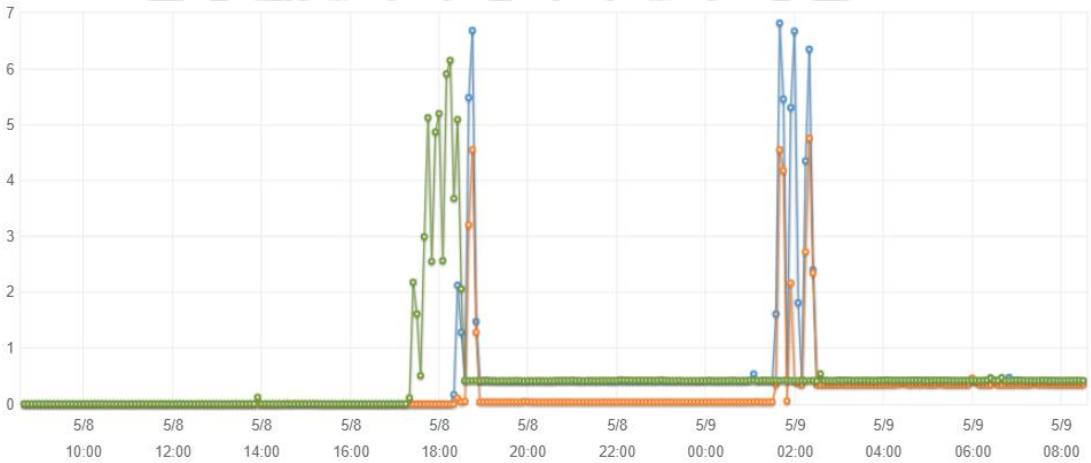


ภาพประกอบ 30 ผลการทดสอบครั้งที่ 2 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker



ภาพประกอบ 31 ผลการทดสอบครั้งที่ 3 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker

จากการทดสอบพบว่าในขณะทดลองการสร้างปริมาณโหลดให้แก่ระบบ Amazon webservices มีการใช้งาน CPU credit usage โดยช่วงเวลาที่ทำการทดสอบเริ่ม 18.00 – 9.35 นาฬิกา โดยแสดงดังภาพประกอบ 32



ภาพประกอบ 32 การใช้งาน CPU credit usage ในขณะทดสอบของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker

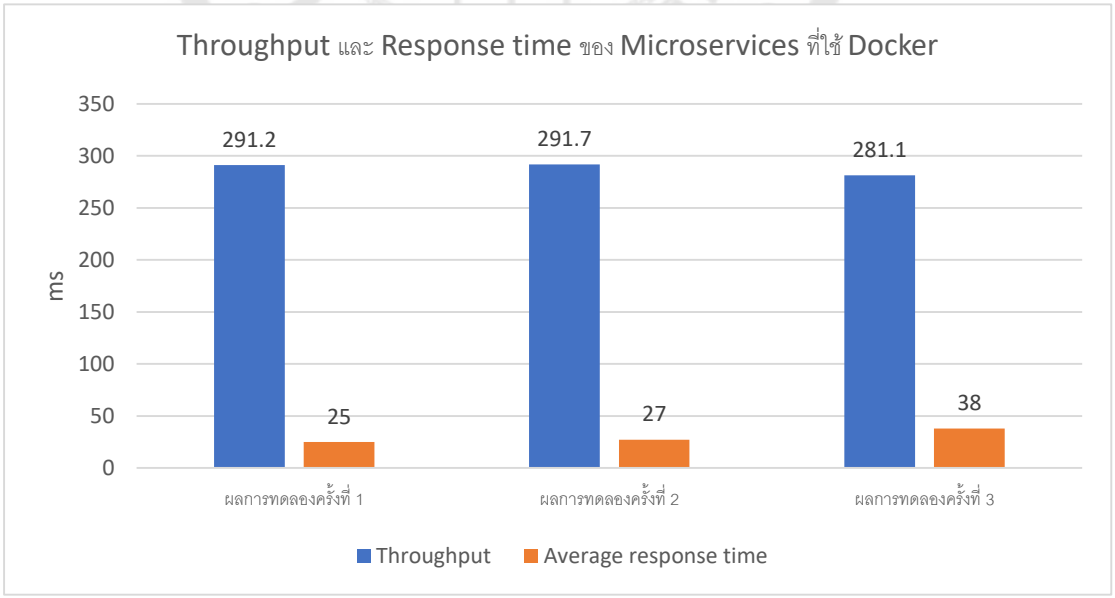
จากการจำลองทดสอบแอปพลิเคชันสะสมแต้มที่พัฒนาด้วยสถาปัตยกรรมแบบ Microservices ด้วย Docker บน Amazon webservices โดยทำการทดสอบโหลด 300

Transactions/second ซึ่งมี Response time เฉลี่ยอยู่ที่ 30 Millisecond และ Throughput 288.00 Transaction/second โดยผลการทดลองทั้ง 3 ครั้งแสดงดังตาราง 7

ตาราง 7 ผลการทดลองของสถาปัตยกรรมแบบ Microservices โดยใช้ Docker

Experimental	Error %	Throughput (tps)	Response time (ms)
ผลการทดลองครั้งที่ 1	0	291.2	25
ผลการทดลองครั้งที่ 2	0	291.7	27
ผลการทดลองครั้งที่ 3	0	281.1	38
เฉลี่ย	0	288.0	30

จากตาราง 7 พบว่า Throughput ของการทดลองแต่ละครั้งมีค่าที่ต่ำกว่า 300 Transaction/second ซึ่งหมายถึงระบบที่ออกแบบด้วย Microservices ส่งผลให้ Throughput ลดลงเมื่อเทียบกับปริมาณการใช้งานที่ 300 Transaction/second ตามที่กำหนดไว้และผลการเปรียบเทียบของการทดสอบแสดงดังภาพประกอบ 29



ภาพประกอบ 33 Throughput และ Response time ของ Microservices ที่ใช้ Docker

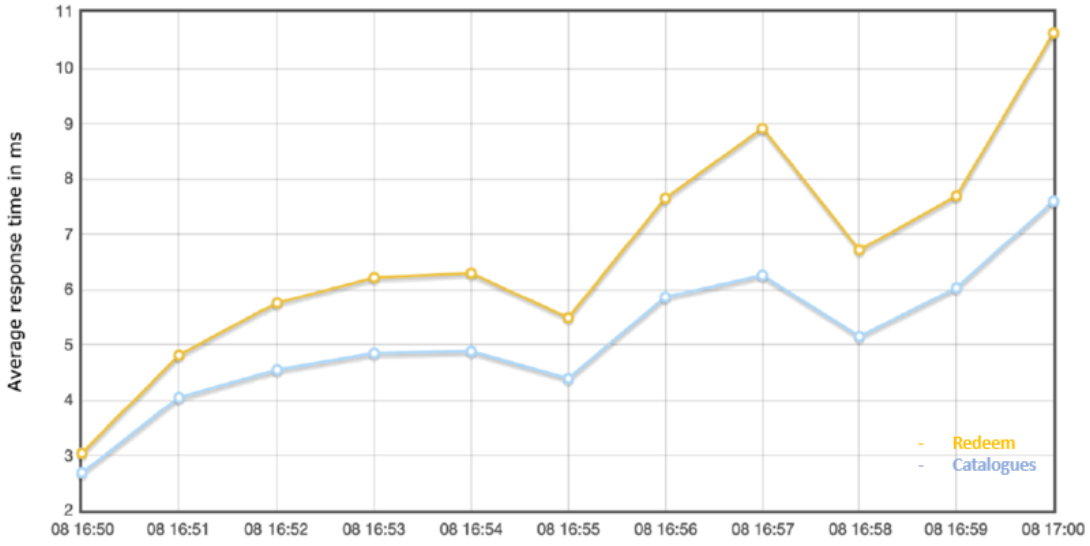
4.3 ผลการทดสอบของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

การทดสอบ Monolith โดยใช้ Kubernetes จำนวน 3 ครั้ง พบว่า Response time และ Throughput ของการทดสอบมีค่าเพิ่มสูงขึ้นและลดลงเนื่องจากจำนวน Thread มีการเพิ่มขึ้นทุกๆ 10 วินาที และมีการใช้งาน Horizontal pod autoscaling ซึ่งจากการทดลองพบว่าในขณะที่ Horizontal pod autoscaling ทำการ replicate จำนวนแอปพลิเคชันนั้นทำให้ Response time ของแอปพลิเคชันนั้นเพิ่มขึ้นโดยจำนวนแอปพลิเคชันแสดงดังภาพประกอบ 34

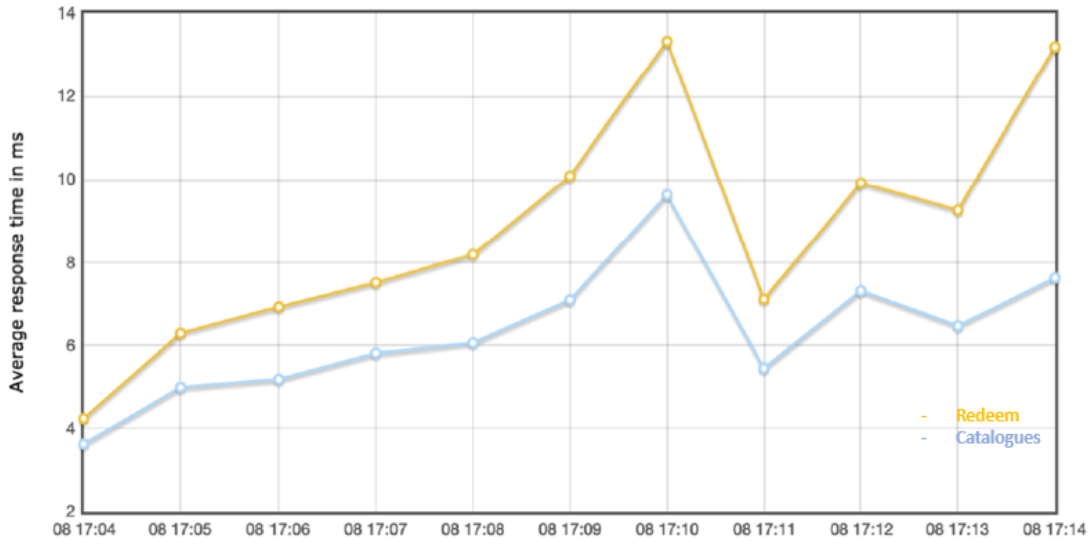
```
NAME          REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
app-monolith  Deployment/app-monolith     45%/50%   1         10        3          10m

NAME          REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
app-monolith  Deployment/app-monolith     39%/50%   1         10        4          10m

NAME          REFERENCE                   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
app-monolith  Deployment/app-monolith     43%/50%   1         10        4          10m
```

ภาพประกอบ 34 จำนวนแอปพลิเคชันที่มีการเพิ่มขึ้นในขณะจำลองการสร้างโหลดบนสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

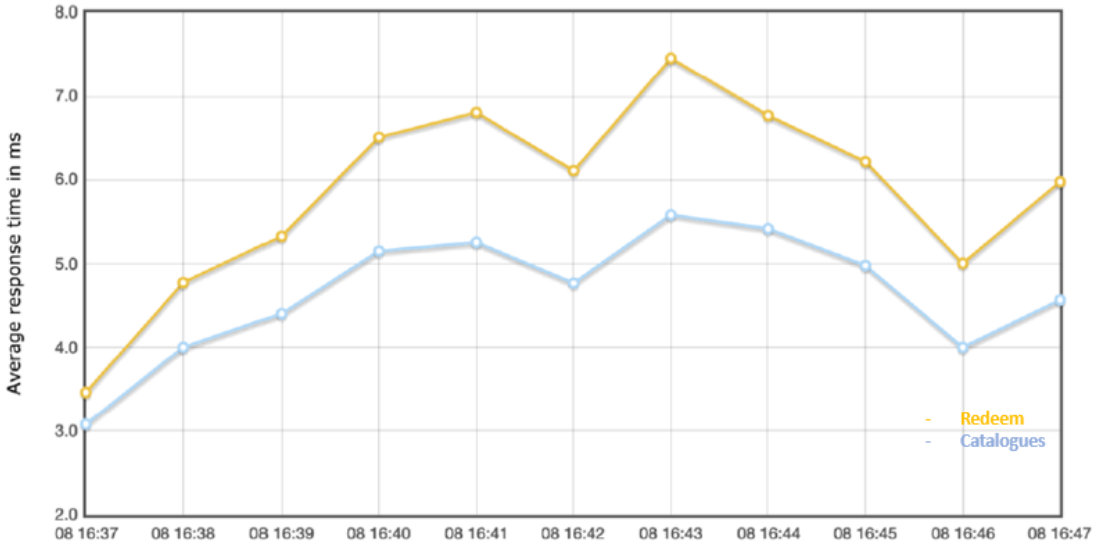
จากทำการ replicate เสร็จสิ้นพบว่า Response time ของการทดสอบทั้ง 3 ครั้ง มีค่าต่ำลงแสดงดังภาพประกอบ 35-37 เมื่อสังเกตุนาทีที่ 3 จะเห็นได้ว่า Response time มีค่าสูงขึ้นและจะลดลงในเวลาต่อมา



ภาพประกอบ 35 ผลการทดสอบครั้งที่ 1 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

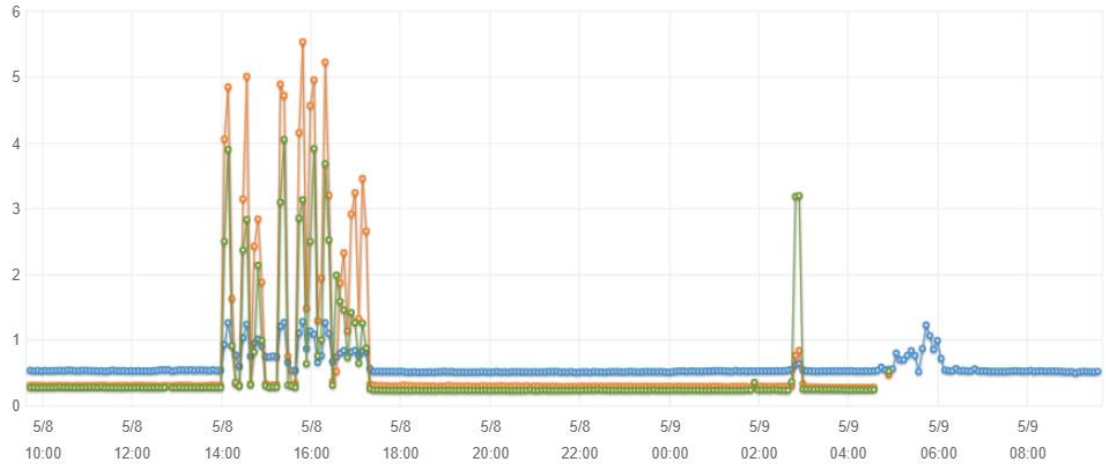


ภาพประกอบ 36 ผลการทดสอบครั้งที่ 2 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes



ภาพประกอบ 37 ผลการทดสอบครั้งที่ 3 ของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

ในขณะทดลองพบว่า Amazon webservices มีการใช้ CPU credit usage เพื่อให้อินสแตนท์ ทำงานได้มีประสิทธิภาพเพิ่มขึ้นโดยช่วงเวลาของการทดสอบจะอยู่ที่ 16.00 – 18.00 นาฬิกาดังภาพประกอบ 38



ภาพประกอบ 38 การใช้งาน CPU credit usage ในขณะทดสอบของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

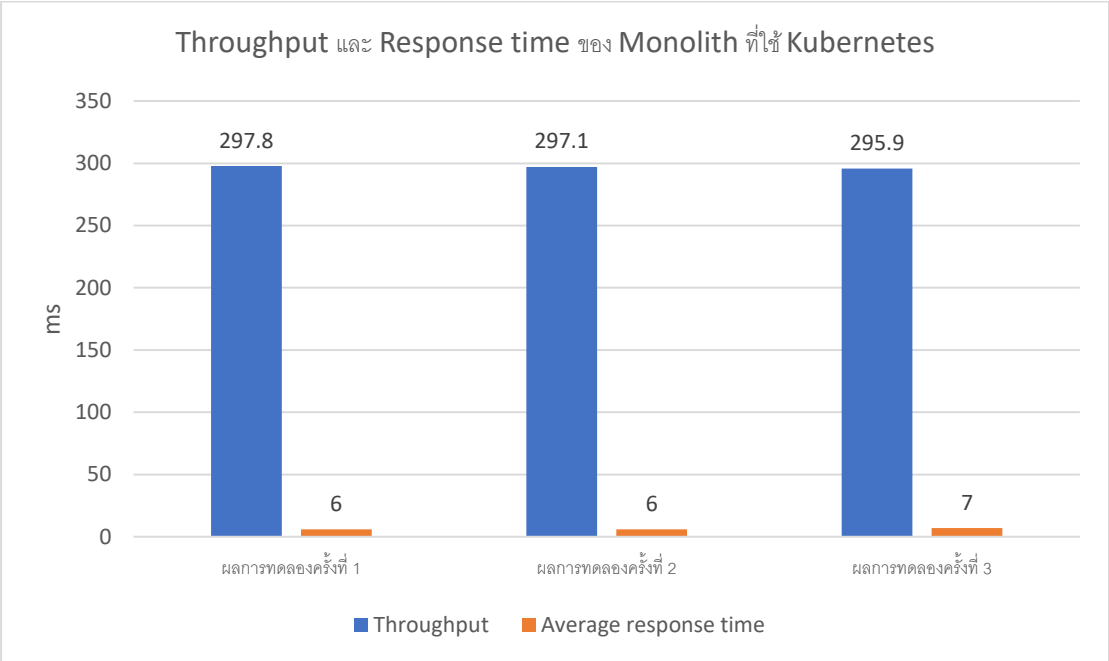
จากการจำลองทดสอบแอปพลิเคชันสะสมแต้มที่พัฒนาด้วยสถาปัตยกรรมแบบ Monolith ด้วย Kubernetes บน Amazon webservices โดยทำการทดสอบโหลด 300 Transactions/second ซึ่งมี Response time เฉลี่ยอยู่ที่ 6.33 Millisecond และ Throughput 296.93 Transaction/second โดยข้อมูลที่ได้จากการทดลองแสดงดังตาราง 8

ตาราง 8 ผลการทดลองของสถาปัตยกรรมแบบ Monolith โดยใช้ Kubernetes

Experimental	Error %	Throughput (tps)	Response time (ms)
ผลการทดลองครั้งที่ 1	0	297.8	6
ผลการทดลองครั้งที่ 2	0	297.1	6
ผลการทดลองครั้งที่ 3	0	295.9	7
เฉลี่ย	0	296.93	6.33

จากตาราง 8 พบว่า Throughput ของการทดลองแต่ละครั้งมีค่าที่ต่ำกว่า 300 Transaction/second เพียงเล็กน้อยซึ่งหมายถึงระบบที่ออกแบบด้วยสถาปัตยกรรมแบบ Monolith โดยมีการทำงานแบบประมวลผลกระจายนั้นสามารถรองรับปริมาณการจำลองโหลดได้

เมื่อเทียบกับปริมาณการใช้งานที่ 300 Transaction/second ตามที่กำหนดไว้และผลการเปรียบเทียบของการทดสอบแสดงดังภาพประกอบ 39



ภาพประกอบ 39 Throughput และ Response time ของ Monolith ที่ใช้ Kubernetes

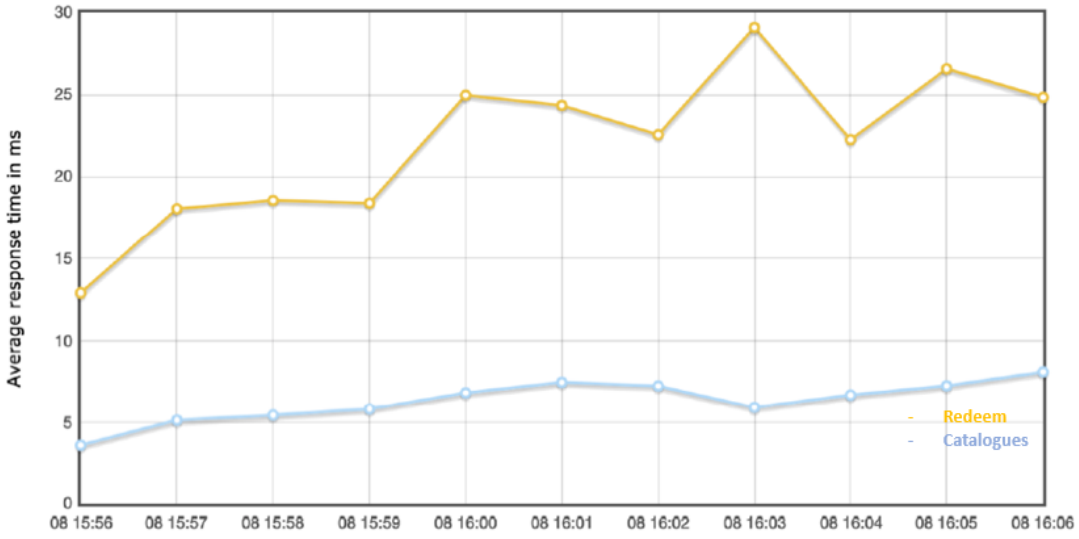
4.4 ผลการทดสอบของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

การทดสอบ Microservices โดยใช้ Kubernetes จำนวน 3 ครั้ง พบว่า Response time และ Throughput ของการทดสอบมีค่าเพิ่มสูงขึ้นเนื่องจากจำนวน Thread มีการเพิ่มขึ้นทุกๆ 10 วินาที และมีการใช้งาน Horizontal pod autoscaling ซึ่งจากการทดลองพบว่าในขณะที่ Horizontal pod autoscaling ทำการ replicate จำนวนแอปพลิเคชันนั้นจะทำให้ Response time ของแอปพลิเคชันนั้นเพิ่มขึ้นโดยจำนวนแอปพลิเคชันแสดงดังภาพประกอบ 40 ตามลำดับ

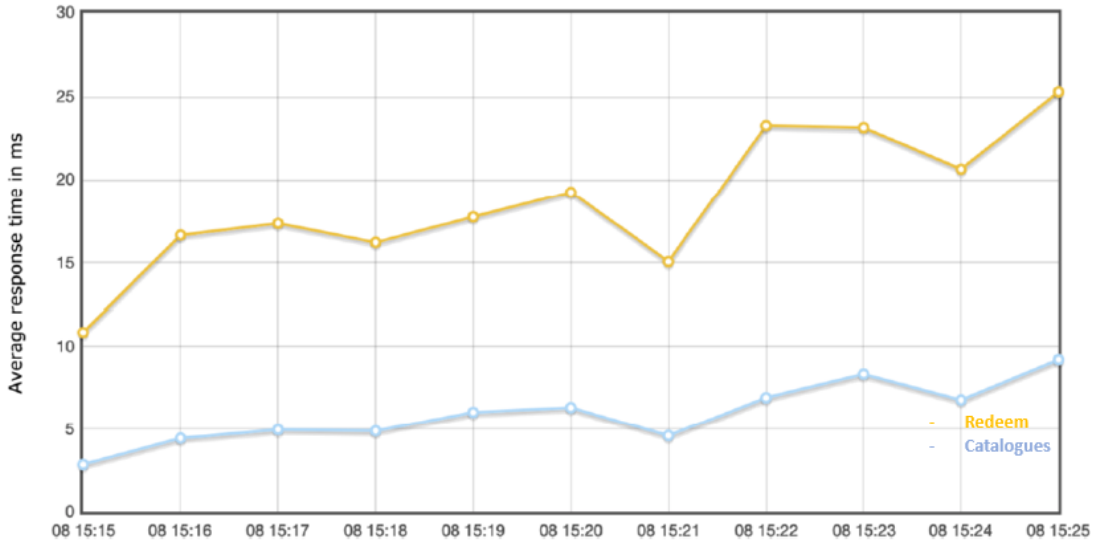
```
NAME                          REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
loyalty-catalogues-service    Deployment/loyalty-catalogues-service     50%/50%   1         10        3          31m
loyalty-redemption-service    Deployment/loyalty-redemption-service     50%/50%   1         10        3          31m
loyalty-users-service         Deployment/loyalty-users-service          30%/50%   1         10        2          31m

NAME                          REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
loyalty-catalogues-service    Deployment/loyalty-catalogues-service     0%/50%    1         10        3          13m
loyalty-redemption-service    Deployment/loyalty-redemption-service     0%/50%    1         10        4          13m
loyalty-users-service         Deployment/loyalty-users-service          0%/50%    1         10        2          13m

NAME                          REFERENCE                                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
loyalty-catalogues-service    Deployment/loyalty-catalogues-service     48%/50%   1         10        3          10m
loyalty-redemption-service    Deployment/loyalty-redemption-service     38%/50%   1         10        4          10m
loyalty-users-service         Deployment/loyalty-users-service          28%/50%   1         10        2          10m
```

ภาพประกอบ 40 จำนวนแอปพลิเคชันที่มีการเพิ่มขึ้นในขณะจำลองการสร้างโหลดบนสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

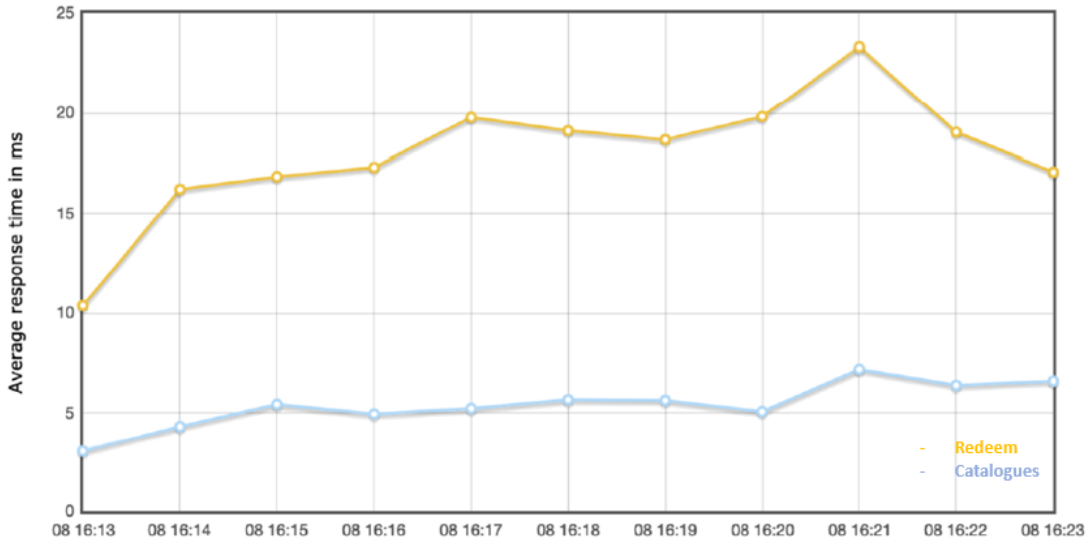
จากภาพประกอบที่ 40 จะเห็นได้ว่าการทดสอบแต่ละ Horizontal pod autoscaling จะทำการ Replicate จำนวนแอปพลิเคชันแตกต่างกันออกไป และเมื่อสังเกตุที่ Targets จะเห็นว่า Horizontal pod autoscaling จัดการ CPU ของแต่ละแอปพลิเคชันไม่ให้เกิน 50% ซึ่งการเพิ่มขึ้นของแอปพลิเคชันจะส่งผลให้ Response time เกิดอาการหน่วงในขณะเริ่ม replicate และผลการทดสอบ Response time ทั้ง 3 ครั้งแสดงดังภาพประกอบ 41-43



ภาพประกอบ 41 ผลการทดสอบครั้งที่ 1 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

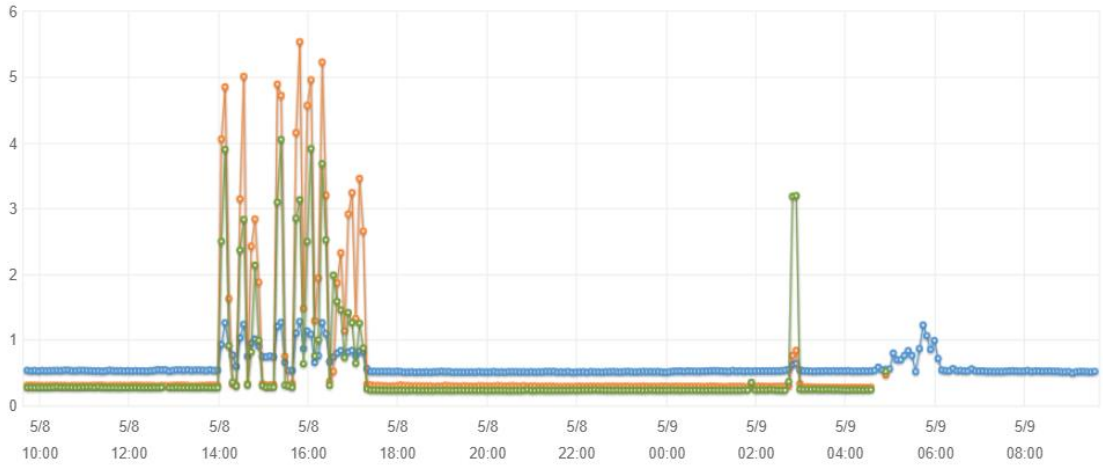


ภาพประกอบ 42 ผลการทดสอบครั้งที่ 2 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes



ภาพประกอบ 43 ผลการทดสอบครั้งที่ 3 ของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

ในขณะทดลองพบว่า Amazon web services มีการใช้ CPU credit usage เพื่อให้อินสแตนท์ ทำงานได้มีประสิทธิภาพเพิ่มขึ้นโดยช่วงเวลาของการทดสอบจะอยู่ที่ 14.00 – 16.00 นาฬิกา ดังภาพประกอบ 44



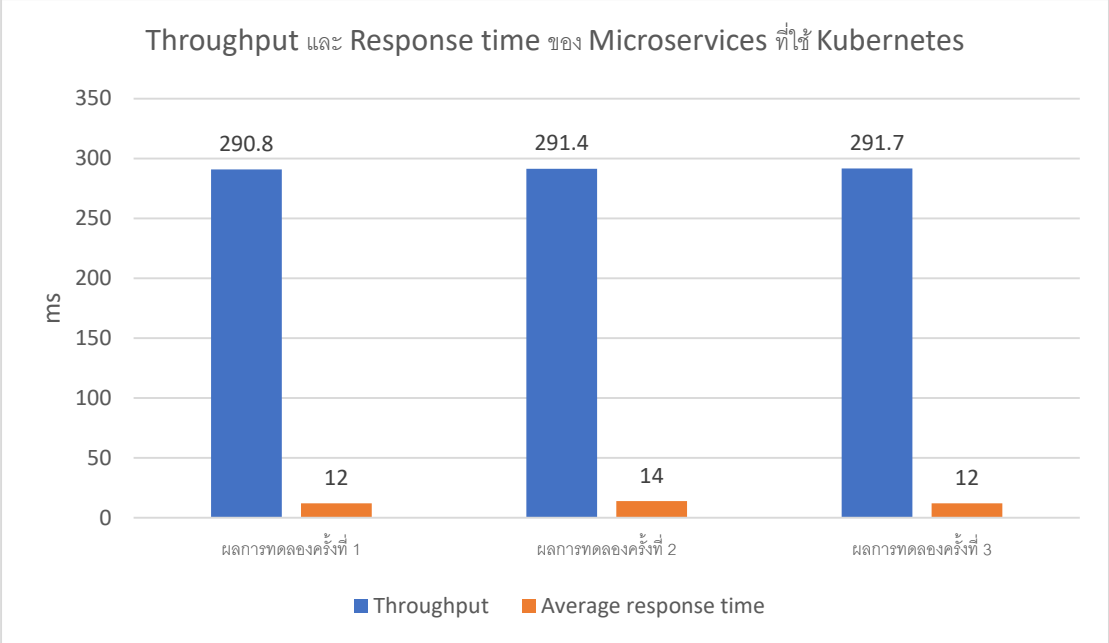
ภาพประกอบ 44 การใช้งาน CPU credit usage ในขณะทดสอบของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

จากการจำลองทดสอบแอปพลิเคชันสะสมแต้มที่พัฒนาด้วยสถาปัตยกรรมแบบ Microservices ด้วย Kubernetes บน Amazon web services โดยทำการทดสอบโหลด 300 Transactions/second ซึ่งมี Response time เฉลี่ยอยู่ที่ 12.67 Millisecond และ Throughput 291.30 Transaction/second โดยข้อมูลที่ได้จากการทดลองแสดงดังตาราง 9

ตาราง 9 ผลการทดลองของสถาปัตยกรรมแบบ Microservices โดยใช้ Kubernetes

Experimental	Error %	Throughput (tps)	Response time (ms)
ผลการทดลองครั้งที่ 1	0	290.80	12
ผลการทดลองครั้งที่ 2	0	291.40	14
ผลการทดลองครั้งที่ 3	0	291.70	12
เฉลี่ย	0	291.30	12.67

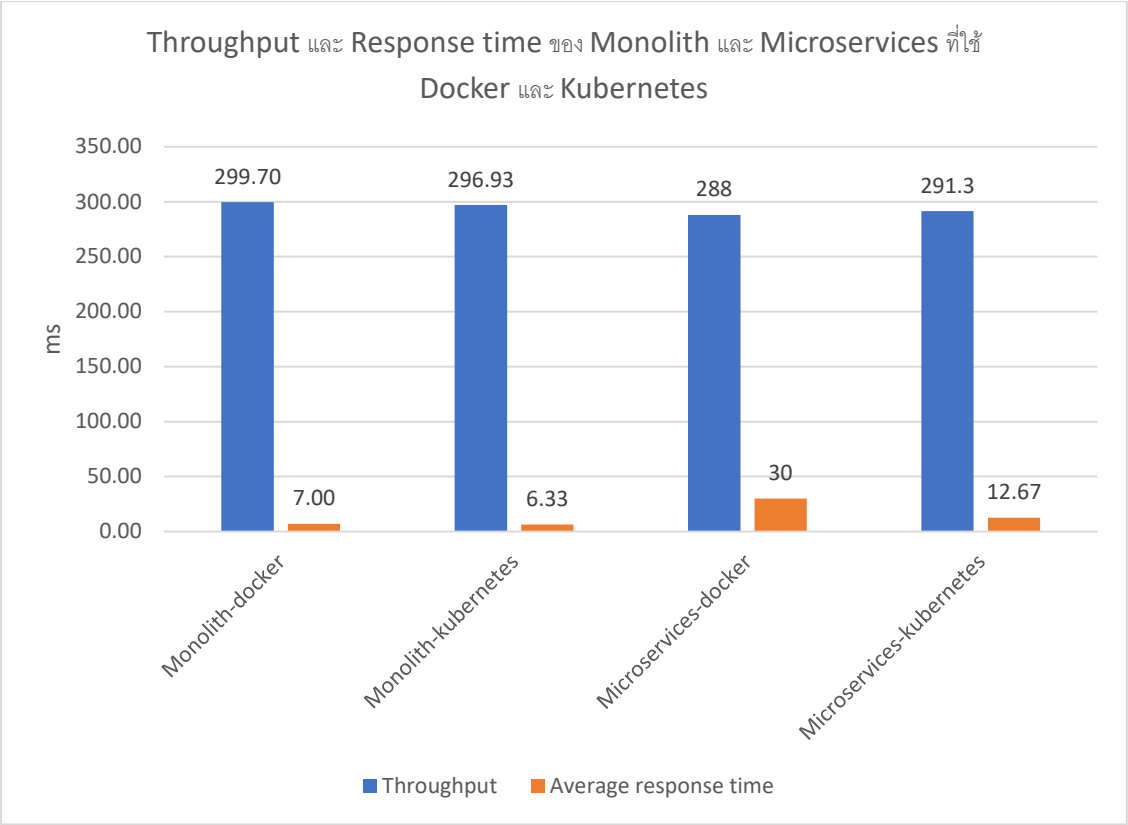
จากตาราง 9 พบว่า Throughput ของการทดลองแต่ละครั้งมีค่าที่ใกล้เคียง 300 Transaction/second ซึ่งหมายถึงระบบที่ออกแบบด้วยสถาปัตยกรรมแบบ Microservices โดยมีการทำงานแบบประมวลผลกระจายนั้นสามารถรองรับปริมาณการจำลองโหลดได้เมื่อเทียบกับปริมาณการใช้งานที่ 300 Transaction/second ตามที่กำหนดไว้และผลการเปรียบเทียบของการทดสอบแสดงดังภาพประกอบ 45



ภาพประกอบ 45 Throughput และ Response time ของ Microservices ที่ใช้ Kubernetes

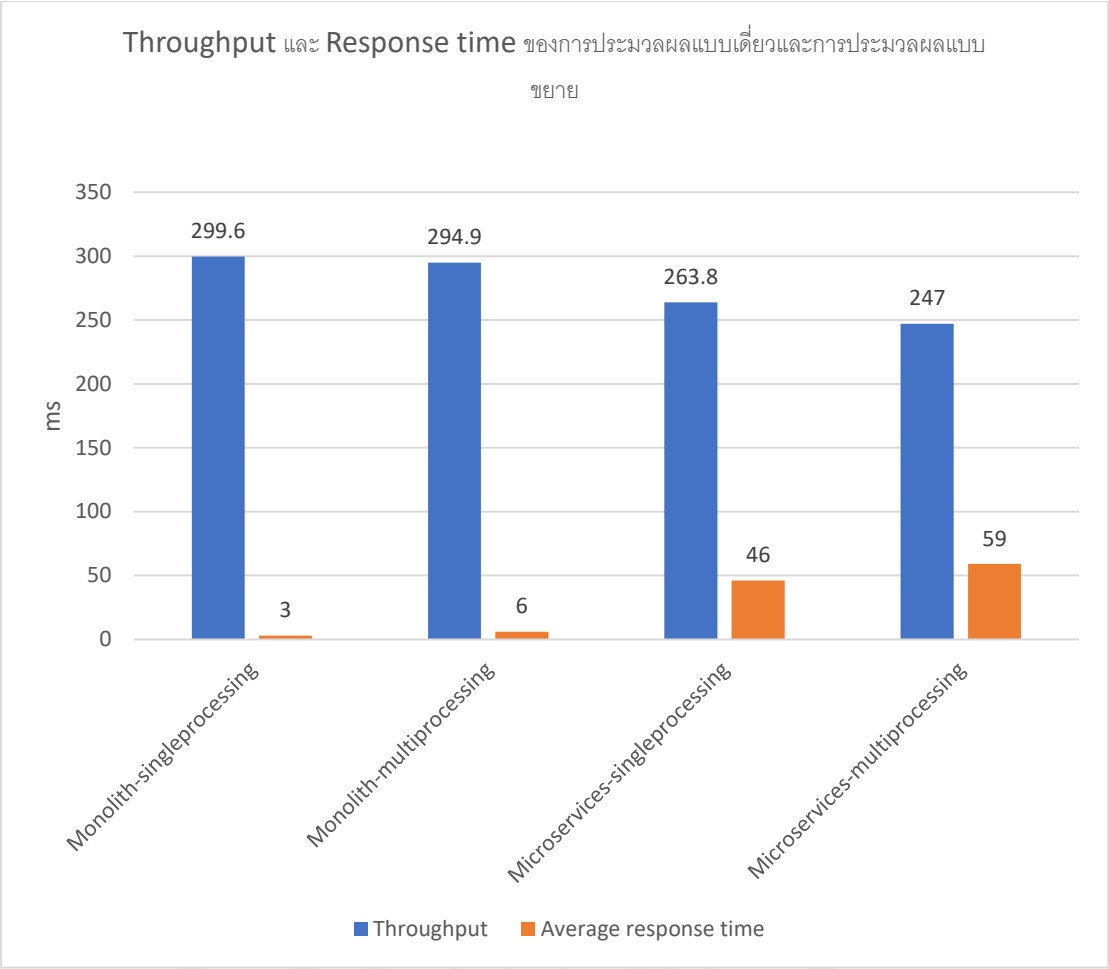
4.5 สรุปผลการทดลอง

จากตารางที่ได้กล่าวมาข้างต้นนั้น เมื่อนำ Response time มาเปรียบเทียบกันแล้วพบว่าสถาปัตยกรรมแบบ Monolith และ Microservices ที่ใช้ Kubernetes ที่รองรับการประมวลผลแบบขยายนั้นมี Response time ที่น้อยกว่าสถาปัตยกรรมแบบ Monolith และ Microservices ที่ใช้ Docker ดังภาพประกอบ 46



ภาพประกอบ 46 การเปรียบเทียบ Throughput และ Response time ของสถาปัตยกรรมแบบต่างๆ

จากแผนภูมิดังกล่าวเห็นได้ว่า Response time ของสถาปัตยกรรมที่ใช้ Kubernetes ในการจัดการกับ Cluster นั้นมีค่าน้อยกว่าการใช้ Docker ผู้วิจัยจึงทำการศึกษาต่อเพื่อหาสาเหตุ โดยทำการเปรียบเทียบการใช้งานของ Kubernetes ใน Cluster ที่มีเพียง 1 Master และ 1 Node ที่มีการทำงานแบบการประมวลผลแบบเดี่ยวและการประมวลผลแบบขยาย หลังจากการทดลองพบว่าการใช้งานการประมวลผลแบบขยายที่มีทรัพยากรจำนวนจำกัดทำให้ประสิทธิภาพลดลง เนื่องจากการขยายตัวของ Horizontal pod autoscaling ที่มีกระบวนการ Warm start ที่ใช้ทรัพยากรจำนวนหนึ่งซึ่งส่งผลให้ประสิทธิภาพการทำงานลดลงเมื่อเทียบกับการประมวลผลแบบเดี่ยว โดยตารางเปรียบเทียบแสดงดังภาพประกอบ 47



ภาพประกอบ 47 การเปรียบเทียบ Throughput และ Response time ของการประมวลผลแบบเดี่ยวและการประมวลผลแบบขยายบน Kubernetes

จากการทดสอบสามารถสรุปได้ว่าการใช้สถาปัตยกรรมใดๆ ในรูปแบบของการประมวลแบบขยายนั้นทรัพยากรที่ใช้ต้องมีความสัมพันธ์กับปริมาณของผู้ใช้เนื่องจากการ Scale out ของ Kubernetes นั้นส่งผลให้ Response time มีค่าสูงเพิ่มขึ้นในขณะที่ Horizontal pod autoscaling มีการสร้างแอปพลิเคชันในรูปแบบ Warm start โดยที่ทรัพยากรส่วนหนึ่งถูกนำไปใช้ในการสร้างแอปพลิเคชัน ซึ่งจากการทดลองยังพบอีกว่าพฤติกรรมของ Horizontal pod autoscaling นั้นจะทำการสร้างแอปพลิเคชันเมื่อเกิด Burst workload ซึ่งส่งผลให้แอปพลิเคชันถูกสร้างขึ้นจำนวนมากและทำให้เกิดปัญหาคอขวดแก่ระบบอันเนื่องมาจากทรัพยากรที่มีจำกัด ยิ่งไปกว่านั้นการทดลองของ Microservices เป็นสถาปัตยกรรมที่มีการประมวลผลแบบกระจายเมื่อใช้ร่วมกับ Kubernetes สามารถช่วยลด Response time ได้

บทที่ 5
สรุป อภิปรายผล และข้อเสนอแนะ

บทนี้นำเสนอผลการเปรียบเทียบประสิทธิภาพของการนำสถาปัตยกรรมแบบ Monolith และ Microservices มาใช้ร่วมกับ Docker และ Kubernetes โดยในบทนี้จะสรุปผลที่ได้จากการทำงานวิจัยและอธิบายเกี่ยวกับข้อจำกัดและข้อเสนอแนะที่จะสามารถนำไปไปพัฒนาต่อยอดได้ในอนาคต

5.1 สรุปผลงานวิจัย

งานวิจัยนี้ได้พัฒนาเพื่อหาประสิทธิภาพของการนำสถาปัตยกรรมต่างๆ มาใช้ร่วมกับเทคโนโลยีสมัยใหม่ เช่น Docker และ Kubernetes เพื่อศึกษาแนวทางและประโยชน์ของการนำไปใช้ต่อยอดในธุรกิจที่ต้องพัฒนาแอปพลิเคชันที่รองรับจำนวนผู้ใช้มหาศาล โดยงานวิจัยนี้ได้ทำการเปรียบเทียบเพื่อหาค่า Response time และ Throughput ของสถาปัตยกรรมและเทคโนโลยีแบบเก่าและแบบใหม่ งานวิจัยนี้ได้แบ่งการทดสอบออกเป็น 4 รูปแบบ ได้แก่ 1) ทดสอบแอปพลิเคชันด้วยสถาปัตยกรรมแบบ Monolith ด้วย Docker 2) ทดสอบแอปพลิเคชันด้วยสถาปัตยกรรมแบบ Microservices ด้วย Docker 3) ทดสอบด้วยสถาปัตยกรรมแบบ Monolith ด้วย Kubernetes 4) ทดสอบแอปพลิเคชันด้วยสถาปัตยกรรมแบบ Microservices ด้วยสถาปัตยกรรมแบบ Kubernetes โดยจำลองการสร้างโหลดขนาด 300 Transactions/second จำนวน 30 Threads โดยเพิ่ม 1 Thread ทุกๆ 10 วินาที ด้วยระยะเวลา 10 นาที เพื่อศึกษข้อดีและข้อด้อยของการนำสถาปัตยกรรมและเทคโนโลยีแบบต่างๆ มาประยุกต์ใช้ร่วมกัน ซึ่งการทดลองในการเปรียบเทียบนี้ต้องคำนึงถึงการออกแบบสถาปัตยกรรม ขั้นตอนการพัฒนาและยิ่งไปกว่านั้นการเลือกคุณสมบัติฮาร์ดแวร์ของให้เหมาะสมกับแอปพลิเคชันเพื่อลดความเอนเอียงซึ่งอาจจะทำให้เกิดข้อผิดพลาดของผลลัพธ์ที่ได้

5.2 ข้อจำกัดของงานวิจัย

จากการทดลองในครั้งนี้พบว่าการผลการทดลองนั้นสามารถคลาดเคลื่อนได้เนื่องความเสถียรของอินเทอร์เน็ต และการทดสอบต้องใช้ Bandwidth ของอินเทอร์เน็ตค่อนข้างสูงจึงไม่สามารถเพิ่ม Throughput ให้มากกว่า 300 Transactions/second ซึ่งอาจจะทำให้เกิดข้อผิดพลาดในการทดสอบได้และการทดสอบในงานวิจัยนี้เป็นเพียงการทดสอบกับแอปพลิเคชันที่ถูกจำลองขึ้นมาและเป็นแอปพลิเคชันที่มีขนาดเล็ก ซึ่งจำนวนโหลด 300 Transactions/second

นั้นเป็นตัวเลขที่กำหนดขึ้นมาเพื่อทดสอบเท่านั้น ในงานวิจัยนี้ไม่ได้ทำการเก็บรวบรวมผลการใช้งานทรัพยากรของฮาร์ดแวร์ซึ่งเป็นข้อมูลที่สำคัญที่สามารถนำมาวิเคราะห์ปัญหาของแอปพลิเคชันได้ และเพื่อให้ผลการทดลองออกมาแม่นยำยิ่งขึ้นการทดสอบนั้นควรเป็นแอปพลิเคชันที่ถูกพัฒนาเพื่อรองรับการใช้งานจริงในธุรกิจและด้วยจำนวนโหลดที่อ้างอิงจากผู้ใช้งานจริงนั้นจะทำให้ได้รับข้อมูลที่แม่นยำยิ่งขึ้น

จากการทดลองจะเห็นได้ว่าสถาปัตยกรรมแบบ Monolith นั้นสามารถทำงานได้อย่างมีประสิทธิภาพมากกว่า Microservices ที่มีการทำงานแบบ Cluster การออกแบบสถาปัตยกรรมนั้นมีข้อดีและข้อจำกัดในหลายๆ ด้านที่แตกต่างกันออกไป โดยงานวิจัยนี้ได้นำเสนอเพียงด้านประสิทธิภาพของการทำงานเพียงด้านเดียวเท่านั้นจึงไม่สามารถเปรียบเทียบได้อย่างชัดเจนว่าสถาปัตยกรรมแบบใดทำงานได้ดีกว่ากันในภาพรวม

5.3 **ข้อเสนอแนะ**

ผู้วิจัยมีความคาดหวังว่าการนำเสนอแนวคิดจากแอปพลิเคชันที่ได้พัฒนานี้จะสามารถนำไปใช้ในการประกอบการตัดสินใจและเป็นประโยชน์ต่อผู้ที่สนใจในด้านการออกแบบสถาปัตยกรรม จึงได้มีการสรุปข้อเสนอแนะที่สามารถนำไปใช้ในการพัฒนาแอปพลิเคชันต่อไปเพื่อให้แอปพลิเคชันมีความยืดหยุ่นและปรับปรุงให้แอปพลิเคชันมีประสิทธิภาพดีขึ้น สามารถระบุรายละเอียดได้ดังต่อไปนี้

1) การพัฒนาแอปพลิเคชันเพื่อให้รองรับการประมวลแบบขยายนั้นสามารถทำให้แอปพลิเคชันมีการทำงานเร็วขึ้นโดยที่ไม่ต้องทำการเพิ่มทรัพยากรคอมพิวเตอร์แต่เมื่อทรัพยากรถูกใช้อย่างเต็มที่แล้วการทำงานแบบดังกล่าวจะส่งผลให้ประสิทธิภาพช้าลง ดังนั้นการวางแผนการและประเมินทรัพยากรถือเป็นเรื่องที่สำคัญ

2) การออกแบบสถาปัตยกรรมที่ดีนั้นต้องออกแบบเพื่อให้ตอบโจทย์ลักษณะของธุรกิจและองค์กรและรองรับการเปลี่ยนแปลงได้ง่าย

3) การนำเทคโนโลยี Container มาใช้แทนเครื่องคอมพิวเตอร์หรือ Virtual machine สามารถช่วยให้การพัฒนาและส่งมอบรวดเร็วขึ้น

บรรณานุกรม

Chung, A., Park, J. W., & Ganger, G. R. (2018). *Stratus: cost-aware container scheduling in the public cloud.* Paper presented at the Proceedings of the ACM Symposium on Cloud Computing.

Cloud, G. (2019). 5 principles for cloud-native architecture-what it is and how to master it. Retrieved from https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it

Delnat, W., Truyen, E., Rafique, A., Van Landuyt, D., & Joosen, W. (2018). *K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters.* Paper presented at the Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems.

Docker. (2019). What is a Container? Retrieved from https://www.docker.com/resources/what-container

Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: up and running: dive into the future of infrastructure*: " O'Reilly Media, Inc.".

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016, 30 March-3 April 2016). *Leveraging microservices architecture by using Docker technology.* Paper presented at the SoutheastCon 2016.

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*: " O'Reilly Media, Inc.".

Kubernetes. (2019). Concepts. Retrieved from https://kubernetes.io/docs/concepts/

Newman, S. (2015). *Building microservices: designing fine-grained systems*: " O'Reilly Media, Inc.".

Ueda, T., Nakaike, T., & Ohara, M. (2016). *Workload characterization for microservices.* Paper presented at the 2016 IEEE international symposium on workload characterization (IISWC).

Vayghan, L. A., Saied, M. A., Toeroe, M., & Khendek, F. (2018, 2-7 July 2018). *Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons*

Learned. Paper presented at the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD).

Vernon, V. (2016). *Domain-driven design distilled*: Addison-Wesley Professional.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015, 21-25 Sept. 2015). *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.* Paper presented at the 2015 10th Computing Colombian Conference (10CCC).

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., . . . Lang, M. (2016, 16-19 May 2016). *Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures.* Paper presented at the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).

Zhao, A., Huang, Q., Huang, Y., Zou, L., Chen, Z., & Song, J. (2019). *Research on Resource Prediction Model Based on Kubernetes Container Auto-scaling Technology.* Paper presented at the IOP Conference Series: Materials Science and Engineering.

ประวัติผู้เขียน

ชื่อ-สกุล	นายนภวิชญ์ ทุมวงษ์
วัน เดือน ปี เกิด	27 เมษายน 2535
สถานที่เกิด	แพร่
วุฒิการศึกษา	พ.ศ. 2554 วิทยาศาสตรบัณฑิต สาขาวิทยาการคอมพิวเตอร์ มหาวิทยาลัยเชียงใหม่ พ.ศ. 2561 วิทยาศาสตร์มหาฑิต สาขาวิชาเทคโนโลยีสารสนเทศ มหาวิทยาลัยศรีนครินทรวิโรฒ