ข้อ 2

ไม่เห็นด้วยกับข้อความข้างต้น เนื่องจาก ปัจจัยหลักที่ต้องคำนึงถึง ที่มีผลต่อการออกแบบสถาปัตยกรรมซอฟต์แวร์ และการตัดสินใจเลือกใช้แนวทางเชิงสถาปัตยกรรม ไม่ได้มีแค่ functional requirements อย่างเดียว แต่มี non-functional requirements ที่รวมถึง Quality Attribute และ Constraints อื่นๆ ด้วย ซึ่ง ที่มีผลกับสถาปัตยกรรมซอฟต์แวร์ส่วนใหญ่ จะมาจาก Quality Attribute เนื่องจากเป็นตัวกำหนดความต้องการของ functional requirement อีกที ว่าทำได้ "ดีแค่ไหน"

ยกตัวอย่างเช่น Quality Attribute ในด้านของ Availability (ความพร้อมใช้งาน) เนื่องจากถ้าเกิดระบบต้องการให้มีการให้มีการพร้อมใช้งานอยู่ตลอด เช่น เมื่อในระบบเครือข่ายมีเครื่องหนึ่งดับไป ก็สามารถให้อีกเครื่องมาใช้แทนกันได้ โดยสามารถเลือกสถาปัตยกรรมอย่างเช่น Peer to Peer (ระบบที่แต่ละ node เชื่อมข้อมูลเข้าหากัน) ถ้ามี node ใด node หนึ่งพัง ก็จะสามารถใช้ node อื่นๆแลกเปลี่ยน หรือแทนข้อมูลกันได้ดี

หรือจะมาจาก Business Goal โดยตรงก็ได้ที่จะเป็นตัวกำหนดเชิงสถาปัตยกรรมโดยตรง เช่น กรณีที่ระบบใช้เป็นไฟล์เก็บข้อมูลได้ปกติ แต่ทางบริษัท อยากให้ทีม database ได้ทำงาน จึงทำให้ต้องคำนึงถึงการใช้งาน DBMS จึงต้องมีการกำหนดสถาปัตยกรรมจาก goal อันนี้โดยตรง

ข้อ 3

โครงการที่มีการออกแบบสถาปัตยกรรมซอฟต์แวร์ ที่ต้องคำนึงถึงมากๆ และมีความสำคัญมากๆ ในการออกแบบ คุณลักษณะจะประกอบไปด้วย

1. Small Solution Space มีทางแก้ที่น้อย และต้องแก้ให้ถูกจุด จึงจะแก้ได้เป็นอย่างดี

2. High Risks มีความเสี่ยงสูง ถ้าเกิดพลาด จะเสียทั้งกำลังคน ทรัพยากร เวลา และเงินที่สูง

3. Quality Attributes ที่มีความละเอียดอ่อน เช่น Security ต้องคำนึงดีๆ ถ้าเกิดผิดพลาด อาจมีความเสียหายสูงได้ เพราะจะทำให้สูญเสียข้อมูลอันสำคัญ หรืออาจโดนโจมตีจากที่อื่นๆได้ง่าย

ยกตัวอย่างโครงการอย่างเช่น

โครงการส่งนักบินอวกาศไปสำรวจดาวอังคาร

จุดประสงค์โครงการ: การส่งจรวดไปดาวอังคาร

เราจึงต้องออกแบบระบบให้เสร็จสับ เป็นอย่างดี ซึ่งจะระบบจะมีความซับซ้อนมาก เพราะจะต้องคำนึงถึงความปลอดภัยของชีวิตของนักบินอวกาศ และการจะส่งจรวดขึ้นไปแต่ละครั้ง จะมีค่าใช้จ่ายที่สูง และใช้กำลังคน และเวลาที่นานมาก ถ้าหากข้อผิดพลาด ก็จะเกิดความเสียหายที่ใหญ่หลวงได้ จึงต้องออกสถาปัตยกรรมให้มั่นใจได้ว่า จะสามารถส่งคนขึ้นไปได้ โดยไม่มีอะไรผิดพลาด

ข้อ 4

เนื่องจากในปัจจุบัน เทคโนโลยีจะอยู่แทบทุก insdustries (อุตสหกรรม) เช่น ในทางการเงิน เศรษฐกิจ ธนาคารในเมื่อก่อนก็จะเป็นการฝากถอนโอนผ่าน ธนาคาร หรือฝากผ่านตู้ ATM แต่ปัจจุบัน เราสามารถโอนเงินให้ผู้อื่นได้ ผ่าน Mobile Application บนมือถือได้ด้วยความง่ายดาย การพัฒนาซอฟต์แวร์เหล่านี้ จึงต้องมีความเร็วในการพัฒนาแอปพลิเคชั่นในระดับหนึ่ง เพื่อตอบสนองความต้องการของผู้ใช้งานที่เพิ่มมากขึ้น และรวดเร็วขึ้นในปัจจุบัน อีกทั้งยังมีคู่แข่งทางด้านบริษัท ที่ทำเกี่ยวกับทางด้านนี้ด้วย จึงทำให้ยิ่งต้องรีบคิด และรีบทำเพื่อตอบโจทย์ลูกค้าให้ไวขึ้น เพื่อเอาชนะคู่แข่งให้ได้

เหตุผลที่1 ว่าทำไมการออกแบบสถาปัตยกรรมจึงมีความสำคัญลดลงไป

คือ ในปัจจุบัน เวลาเราจะพัฒนาอะไร จะมีรูปแบบ Pattern ที่มีมาให้อยู่แล้ว เช่น Framework ของการพัฒนา frontend Application (vue, react, angular) ที่จะมี pattern/style เป็นแบบ MVC เพื่อที่จะ match กับการเปลี่ยนแปลงกับหน้า UI ที่เปลี่ยนแปลงบ่อย และรองรับการเกิด interaction ต่างๆได้ดี ซึ่ง MVC จะสามารถทำตรงนี้ได้ดี ซึ่งเป็นเหมือนสิ่งที่เค้าคิดมาให้อยู่แล้ว เราสามารถหยิบนำไปใช้ได้เลย ด้วยความรวดเร็วนี้ เราก็ไม่จำเป็นที่จะต้องมาออกแบบ สถาปัตยกรรมใหม่อีกครั้ง เพียงแค่ทำความเข้าใจถึงสถาปัตยกรรมที่นำมาหยิบใช้ และใช้มันให้ถูกต้อง

เหตุผลที่ 2

คือ ปัจจุบัน Application ที่อาศัยความรวดเร็วนี้ อาจไม่จำเป็นที่จะต้องคิดสถาปัตยกรรมอย่างจริงจัง เนื่องจากใช้ระยะเวลาที่นาน และในปัจจุบัน จะเน้นการทำงานแบบ Agile (ให้ความสำคัญกับลูกค้าก่อน Highest customer value first) จึงทำให้จะเป็นการเน้นทำไปเรื่อยๆ และมีการ evolutionary design (design ไปเรื่อยๆ พร้อมทำไปด้วยเป็นขั้นๆ) เพื่อให้ได้ product ให้ไวที่สุด เพื่อที่จะสนองความต้องการได้ไว และแข่งกับคู่แข่งได้ดี

ข้อ 5

การคำนวณ Availability อันดับแรกจะมี 2 อย่างหลักๆ

1. ถ้าเกิดการเชื่อมต่อเป็นแบบอนุกรม (Serial) เช่น WebServer -> AppServer -> DBServer

จะต้องใช้การคำนวณแบบ คูณกัน คือเอา Avaliability ของแต่ละส่วน มาคูณกัน จะได้คำตอบออกมา

2. ถ้าเกิดการเชื่อมต่อเป็นแบบขนาน (Parallel) เช่น AppServer 01,02,03 (เลือกอย่างใดอย่างหนึ่ง)

จะต้องใช้การคำนวณแบบ นำ ค่า 1 - Avaliability ของ AppServer = X (ค่าความน่าจะเป็นที่จะ down)

จากนั้น นำ X มา คูณกันตามจำนวนเครื่องที่ พาราเรลกัน (ในที่นี้เป็น 3 เครื่องของ AppServer) จะเป็น X\*X\*X = parallelX (ค่ารวม)

จากนั้น สุดท้าย นำค่า 1 มาหักลบออก parallelX จะได้เป็น 1 - parallelX = ค่า Avaliability Parallel นั้น ออกมา

การคำนวณจะเป็น

Web(1) -> App(3) -> DBServer (2)

อันดับแรก หา Avaliability Parallel ของ App กับ DBserver ก่อน

จะได้ว่า

App = 90.0 %

1-0.9 = 0.1

นำ 0.1 \* 0.1 \* 0.1 = 0.001

นำ 1 - 0.001 = 0.999 หรือ 99.9 %

DBserver =95.0 %

1-0.95 = 0.05

นำ 0.05 \*0.05 = 0.0025

นำ 1 - 0.0025 = 0.9975 หรือ 99.75 %

จะได้ว่าสุดท้าย

Web(1) -> App(3) -> DBServer (2)

99.9 % \* 99.9 % \* 99.75 %

จะได้เท่ากับ

0.999 \* 0.999 \* 0.9975 = 0.9955059975

หรือ 99.55059975 %

หรือ 99.5506 %

ข้อ 6

เห็นด้วย (แต่ไม่ใช่ทั้งหมด)

ในส่วนที่เห็นด้วย จริงๆความหมายของ interoperable คือการตีความหมายที่ตรงกัน / format ตรงกัน / เข้าใจและสามารถแลกเปลี่ยนข้อมูลกันได้ การที่ทางองค์กรหรือหน่วยงานกลางจัดทำ Protocol (กฏของภาษา) ให้ทุกคน follow และปฏิบัติตาม ก็เป็นหนึ่งใน interoperable ที่ทำให้เมื่อใช้ Protocal เดียวกัน ก็จะสามารถสื่อสาร และเข้าใจความหมายของอีกฝั่งได้

แต่...

เพราะจริงๆ การที่ interoperable ได้ อาจไม่จำเป็นต้องใช้ Protocol เพื่อกำหนดให้ทั้งสอง

ต้องใช้มาตราฐานเดียวกันเพียงอย่างเดียวเสมอไป ยังมีวิธีการอื่นที่ทำให้เกิด interoperable ได้

ยกตัวอย่างเช่น สมมติหน่วยงาน A และ หน่วยงาน B แต่ละคนใช้คนละภาษากัน โปรแกรมคนละแบบ แล้วถ้าหาก หน่วยการ A ต้องการจะติดต่อที่หน่วยงาน B จะต้องใช้ตัวกลางหนึ่งตัว เช่น Broker (สถาปัตยกรรมซอฟแวร์แบบหนึ่ง) จะเป็นเหมือนตัวแปลงข้อมูลให้สามารถให้หน่วยงาน A และหน่วยงาน B สามารถแลกเปลี่ยนข้อมูลกันได้ โดยไม่จำเป็นต้องเป็นภาษา, โปรแกรม หรือ Protocol ตัวเดียวกัน

ข้อ 7

cohesion ต่ำ มีผลทำให้ module หรือ class นั้นๆ มี method ที่ไม่เกี่ยวข้องกันเยอะ เช่น สมมติ class เกี่ยวกับการเงิน ก็ควรจะเป็น method การฝาก ถอน โอนเงิน และการทำธุรกรรมอื่นๆ แต่ถ้าเกิดมี method ที่ไม่เกี่ยวข้องเช่น มีเรื่อง login หรือเก็บรายชื่อ (ไม่ได้เกี่ยวกับการเงิน) ก็จะทำให้ cohesion ลดต่ำลงภายใน module นั้นๆ

ทำให้เวลาจะแก้ในภายหลัง จะทำให้ต้องมาแก้ภายใน 1 module อาจจะไปกระทบกับ method อื่นๆภายใน module นั้นด้วย ซึ่งเป็นสิ่งที่ไม่ดี (ควรแก้แล้วจบที่ตรงนั้น ไม่ควรไปกระทบส่วนอื่นๆใน 1 module)

coupling สูง คือมี module หรือ class มากกว่า 2 ขึ้นไป ที่มีความเกี่ยวเนื่องกัน เช่น สมมติมี 3 modules ถ้าเกิดมีการแก้ไข module ที่ 1 อาจจะต้องไปแก้ไขใน module ที่ 2 และ 3 ด้วย ทำให้เวลาแก้ 1 ที ต้องแก้ถึง 3 modules ทั้งๆที่จริงๆ เราต้องการจะแก้แค่ module แรก เพียง module เดียว

เพราะฉนั้น การที่จะตอบโจทย์ modifiability ที่ดีได้ จะต้องทำให้ cohesion สูง (ภายใน 1 module มีความเกี่ยวข้องและสัมพันธ์กัน) และ coupling ต่ำ (ทำให้มีหลาย module ไม่เกิดการเชื่อมโยงกันมากเกินไป หรือโดยไม่จำเป็น) จะทำให้การแก้ไขในภายหลัง ทำได้ง่ายมากขึ้น

ข้อ 8

จากโจทย์ข้อนี้ สิ่งที่ได้เป็น Quality Attribute หลักๆ ออกมามีดังนี้

1. Interoperablity (การทำงานร่วมกันได้)

2. Modifiablility (การแก้ไข/ปรับปรุง/เปลี่ยนแปลง ทำได้ง่าย)

เลือก Architectural pattern/style ที่เหมาะสมจะมี สองอย่างด้วยกัน คือ

1. Service-Oriented Architecture (SOA)

เพราะ SOA เป็น architect ที่ควบรวม Services หลายๆตัวเข้าด้วยกันได้ โดยจะเป็นตัวกลางที่ให้รวมแต่ละ Service ไว้ด้วยกัน และใครอยากใช้ ก็สามารถเข้ามาใช้งานได้ ผ่านจุดศูนย์รวม Service นี้ (โดยปกติจะเป็น ESB หรือ Enterprise Service Bus) และมี Orchestrate (ประสาน) ให้ service สามารถมาใช้งานร่วมกันได้เยอะๆ

โดย SOA จะตอบโจทย์ทั้ง Interoperability และ Modifiablity

ในด้าน Interoperability คือคนที่เรียกใช้งาน Service ก็ไม่จำเป็นต้องรู้รายละเอียด Service นั้นๆเยอะ ขอแค่ติดต่อไปที่ตัวกลางได้ ก็สามารถแลกเปลี่ยนข้อมูลกันได้แล้ว

ในด้าน Modifiability คือจะเป็น dynamic reconfigulation in ESB ก็คือสามารถปรับแต่ง Service ได้โดยไม่มีผลกระทบในตัว ESB ใหญ่ๆ

สิ่งที่ต้องระวังคือ ระบบจะมีความซับซ้อนมากๆ เนื่องจาก services มีเยอะ และอาจเกิด bottle neck ของการใช้ service ได้ และอาจทำให้ Performance ลดลง (เกิดจาก Overhead ในการส่ง)

2. Broker

เพราะ Broker เป็น architect ที่เป็นตัวกลาง ตัวแทน

Broker จะเป็นเหมือนตัวกลางที่จะเชื่อมระหว่าง Client และ Server เข้าด้วยกัน โดยจะควบรวม Services เข้าด้วยกัน

ในด้าน Interoperability คือ ในการสื่อสารระหว่าง Client กับ Server โดย Client ไม่จำเป็นต้องรู้ว่า Server อยู่ที่ไหน หรือ Server ก็ไม่จำเป็นต้องรู้ว่า Client อยู่ที่ไหน ก้ให้ Broker เป็นตัวจัดการแลกเปลี่ยนข้อมูลกันภายในตัว Broker ได้

ในด้าน Modifiability คือสามารถปรับเปลี่ยนตัว broker ได้ง่าย

สิ่งที่ต้องระวังคือ อาจเกิดความซับซ้อน และอาจเกิด latency ได้ ระหว่างการส่ง (เกิดจาก Overhead)

ข้อ 9

จากโจทย์ข้อนี้ สิ่งที่ได้เป็น Quality Attribute หลักๆ ออกมามีดังนี้

1. Scalability (สามารถขยายได้ดีจากหลากหลายสาขาจำนวนมาก)

2. Avaliability (ถ้ามี node ใด node หนึ่งพังไป จะยังสามรถทำงานได้อยู่ (ในMap reduce))

3. Performance (รองรับ bandwidth ที่เยอะ และต้องประหยัด bandwidth)

เลือก Architectural pattern/style ที่เหมาะสมจะมี สองอย่างด้วยกัน คือ

1. Shared-Data (Data-centric Architecture)

เพราะ การที่จะมีการเก็บรวบรวม Log file ที่มาจากหลากหลายบริการ มาไว้ในที่ๆเดียวกัน จะเหมาะสมกับ Shared-Data มากๆ เพราะ Shared-Data จะมีจุดศูนย์รวม ที่จะคอยเก็บรวบรวมข้อมูลต่างๆ จากหลากหลายแหล่ง เพื่อให้หลาย Client สามารถเก็บ log file ได้ และให้ Client คนอื่นๆ(องค์กรอื่นๆ) สามารถ access เข้ามาดึงข้อมูลจากจุดศูนย์กลางตรงนี้ได้เหมือนกัน

2. Map-Reduce

เพราะ เหมาะสำหรับรับข้อมูลปริมาณมากๆ ซึ่งจะต้องมีการประมวลผล หรือการนับคำ ออกมาเป็น field ต่างๆ ตรงนี้จะเหมาะกับ Map-Reduce มาก ซึ่งการทำงานก็คือจะมีหลาย node ที่รับข้อมูลมาแล้วมาประมวลผลและส่งต่อให้ node ต่างๆต่อไปเพื่อทำให้เกิด Avaliablity(พังไป 1 node ก็ยังทำงานได้) และ Performance ( bandwidth เมื่อสามารถรองรับ data ที่เยอะมากๆจากหลายๆสาขา)

เวลาใช้จริง เราต้องใช้ Shared-Data และ Map-Reduce ร่วมด้วยกัน เพื่อตอบโจทย์สถานการณ์นี้

ข้อ 10

จากโจทย์ข้อนี้ สิ่งที่ได้เป็น Quality Attribute หลักๆ ออกมามีดังนี้

1. Modifiablity

เลือก Publish-Subscribe น่าจะเหมาะสมที่สุด

เนื่องจาก Publish-Subscribe เป็น architect ที่รองรับการทำ message queue โดยมีตัว Message Oriented Middleware เป็นตัวกลางที่คอยจัดการข้อมูลต่างๆ เป็น Topic ๆ (ซึ่งในสถานการณ์นี้ จะเป็น dashboard ที่แต่ละทีมจะมีข้อมูลเฉพาะของตนเอง) ซึ่ง ถ้าเกิด Subscriber ต้องการที่จะติดตามดูใน dashboard(หรือ Topic ใดๆ) ก็จะติดตามในแต่ละ dashboard หรือ Topic นั้นๆไว้ และ Publisher ถ้าต้องการที่จะประกาศข้อมูล message ต่างๆ ก็แชร์หรือส่งข้อมูลไปให้ Topic ที่เลือก เพื่อให้ Subsriber รอรับข้อมูลได้

ที่สำคัญคือ Subscriber สามารถมีหลายคนได้ ในการติดตามแต่ละ Topic (ในกรณีนี้อย่างเช่น CFO และ CMO ก็ต่างสนใจข้อมูลรายได้จากการขายสินค้าเหมือนกัน ก็ติดตาม Dashboard เดียวกัน) หรือจะ ติดตาม topic อื่นๆ มากกว่า 1 อันได้

รวมถึง Publisher ด้วยเช่นกัน ที่สามารถ มีได้หลายๆคน เพื่อ publish ข้อมูลต่างๆลงไปใน topic หรือ dashboard ที่สร้างขึ้น

ถ้าเกิดมีการเปลี่ยนแปลง (ในกรณีนี้จะเป็น "องค์การอาจจะปรับใช้ระบบ CRM ใหม่ ซึ่งเป็นการเพิ่มแหล่งข้อมูลใหม่เข้าในระบบ") ก็จะตอบโจทย์ Modifiablity เนื่องจาก ถ้ามีส่วนอื่น หรือ Topic/Subsriber/Publisher มาเพิ่ม ก็สามารถเพิ่มเข้าไปได้เลย จะไม่กระทบเยอะกับตัวหลัก หรือที่มีอยู่ก่อนแล้ว ซึ่งเป็นการง่ายต่อการเพิ่มและเปลี่ยนแปลงในภายหลัง

ข้อ 11

metrics ตัวชี้วัด ในเชิง Performance ดังนี้

1. Throughput (ขนาดการส่งข้อมูล)

ใช้เพื่อรองรับจำนวนผู้สมัครที่มีปริมาณมาก ถึง 100000 คนต่อวินาที ขนาดการส่งข้อมูลจะเยอะมาก จะต้องรองรับตรงนี้ได้ โดยข้อมูลควรรับได้ จากปริมาณข้อมูลที่กำหนดไว้ต่อวินาที เช่น หากมีคนใช้งาน 100000 คนต่อวินาที เรียกข้อใช้ข้อมูลพร้อมกัน เซิฟเวอร์เว็บไซต์จะมีการส่งข้อมูลได้อย่างน้อย 200 MB ต่อการ request 1 ครั้ง

2. Data Latency/ Response Time (เวลาตอบสนอง)

ทั้งเรื่องของการล้าช้า หลังจากที่ผู้ใช้มีการตอบสนอง และมีการส่งข้อมูลกลับมา จะต้องคำนึงถึงค่า latency, jitter เพื่อให้มีค่าไม่สูงเกินไป และมีการตอบสนองในเวลาที่กำหนด เช่น หลังจากที่ผู้ใช้งานร้องขอ/ เรียกหน้าเว็บไซต์ จะต้องมีการส่งข้อมูลหน้าเว็บไซต์มาภายในระยะเวลา 2 วินาที ในสภาวะปกติ หรือ 10 วินาที ในสภาวะเซิฟเวอร์ Overload

ยกตัวอย่างสถานการณ์

สถานการณ์ : Web/Application Server สามารถรองรับคนใช้งาน/1วินาที ได้สูงสุด 120000 คน

ลงทะเบียนระยะแรกมีจำนวนผู้สมัครมาลงทะเบียนสูงถึง 100000 คนต่อวินาที

Scenario ->

Source of Stimulus (แหล่งที่มา): ผู้ใช้ระบบลงทะเบียน (Client)

Stimulus (สิ่งที่เกิด): Request จำนน 100000 คนต่อ 1 วินาที

Artifact (สิ่งที่สนใจ): Web/Application Server

Environment (สภาพแวดล้อม): การทำงานในสภาวะปกติ (ไม่เกินขีดจำกัดที่รับได้)

Response (ตอบสนอง): เซิฟเวอร์มีการตอบสนอง และส่งข้อมูลกลับไปหาผู้ใช้งาน และตรวจสอบจำนวนคนที่เข้าใจงาน

Response measurement (วัดผล): ตรวจพบการเข้าใช้งานจำนวน 100000 คนต่อ 1วินาที และต้องมีการส่งข้อมูลกลับไปหาแต่ละ Client ได้อย่างน้อย 200 MB ต่อ 1 request ภายในระยะเวลา 2 วินาที

ข้อ 12

Evaluation Phase 1 ตามวิธี ATAM

1. Present The ATAM

ในขั้นแรก Evaluation Leader (ที่ไปว่าจ้างจากภายนอก) จะนำเสนอวิธี ATAM ให้กับ Project Representative (เช่น Project Manager, Project Sponser รวมถึง ทีม Architect ด้วย) เพื่อให้ทุกคนได้เข้าใจวิธีการของ ATAM และกำหนดบทบาท และสิ่งที่คาดหวังหลังจากการทำ ATAM สำเร็จ (อธิบายเป็น Step และบอก Output ว่าจะได้อะไรจาก ATAM บ้าง)

2. Present The Business Driver

Business Driver หรือแรงผลักดันทางธุรกิจ

ในขั้นที่สอง Project Representative จะมา present ตัวระบบที่จะทำโดยสังเขป จากมุมมองของ Business view

โดยรายละเอียดมีดังนี้

- นำเสนอ function ที่สำคัญๆ

- ข้อจำกัด เช่น ทาง technical, เศรษฐกิจ, บริหาร หรือนโยบายอื่นๆที่เป็นข้อจำกัดของโปรเจคนี้

- Business goal เป้าหมายทางธุรกิจ

- Major stake holders คนสำคัญหลักๆที่เกี่ยวกับข้องโปรเจคนี้

3. Present The Architecture

ในขั้นที่สาม ในฝั่งของ Team Architecture (ภายในองค์กร) , Lead Architect จะนำเสนอ architecture ในระดับที่เหมาะสม จากที่ได้คิดมาก่อนแล้ว และนำเสนอข้อจำกัดแบบมีรายละเอียดลึกลงไปเช่น ข้อจำกันข้อ ระบบ os, hardware, middleware หรืออื่นๆที่จะ interact กับตัว project นี้

4. Identify Architectural Approaches

ในขั้นที่สี่ Evaluation Team จะเริ่มทำการประเมินโดยหา tactic ที่สำคัญๆ Pattern ที่ทางทีม architect นำเสนอมา มากลั่นกรอง และหา Quality Attribute ที่ซ่อนอยู่ และสำคัญๆ ออกมา ซึ่งตลอดการทำ จะมี Scribe (ฝั่ง Evaluation team) คอยจดข้อมูลทั้งหมดทั้งบน board และ บน electronic form

5. Generate Quality Attribute "Utillity Tree"

ในขั้นที่ห้า Evaluation Team จัดทำ ASR ให้เป็นรูปธรรม โดยร่วมเขียนกับ Project Representative

ASR จะเป็น Architecture Significant Requirement ที่มีความสำคัญมากที่มีผลต่อระบบโครงสร้างของสถาปัตยกรรมที่เลือก

6. Analyze Architectural Approaches

ในขั้นที่หก Evaluation Team เอา Scenario ที่สำคัญที่สุด มาไล่ตรวจสอบไปเรื่อยๆ โดยเน้นความสำคัญที่ มี Requirement อะไรบ้าง และเราได้ตัดสินใจอะไรไป Quality Attribute หรือ ASR ที่คิดมาตอบโจทย์ยังไงกับ Business goal หรือ architect ที่เลือก แล้วมีข้อดีข้อเสียยังไงที่ควรต้องรู้ (Trade off)