Serverless Computing in Enterprise Application Integration: An Organizational Cost Perspective

Javier Leung

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2021

Bachelor of Business Administration

Degree Programme in Business Information Technology

Abstract

Author
Javier Leung

Degree
Bachelor of Business Administration, Business Information Technology

Thesis title
Serverless Computing in Enterprise Application Integration: An Organizational Cost Perspective

Number of pages and appendix pages
84 + 8

Serverless computing has become increasingly popular as a deployment model for microservices. However, not much has been written about its applications in enterprise application integration (EAI), namely the practice of sharing data, business rules and processes between independently designed applications within an organization. Moreover, there is yet little consensus on what constitutes an optimal EAI architecture, making it difficult to evaluate the efficacy of serverless in its context.

Using the design science methodology, this research aims to (i) develop an abstract model for understanding EAI architecture, and (ii) evaluate the efficacy of serverless computing in EAI based on this model. Drawing from research in agile software development, organizational structure (U-form and M-form organizations) and transaction cost economics (firm boundaries, markets), we establish a theoretical framework in which EAI architectures can be understood as *vertical hierarchies* and *horizontal networks* involving tradeoffs between technological and organizational factors.

From this framework, we first define an abstract model of EAI architecture as a structure that seeks to economize on the production, coordination, and vulnerability costs of organization, while minimizing transaction costs incurred from search, decision-making, and enforcement. Second, we describe the main EAI architectural patterns in terms of our abstract model and examine how serverless emerges from this context. Lastly, using a hypothetical business case in which several information systems applications must be integrated behind a customer-facing channel, we design and implement a serverless integration runtime using Amazon Web Services and evaluate its efficacy as an integration architecture based on our model.

Keywords

Serverless Computing, Integration Patterns, Enterprise Architecture, Cloud Computing, Organizational Structure, Transaction Cost Economics

Table of contents

Abbreviations

API	Application Programming Interface
ARN	Amazon Resource Names
AWS	Amazon Web Services
B2C	Business to Consumer
BI	Business Intelligence
BPM	Business Process Management
capex	Capital Expenditure
CBA	Cost-Benefit Analysis
CDK	Cloud Development Kit
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command Line Interface
CM	Content Management
CPU	Central Processing Unit
CRM	Customer Relationship Management
CSV	Comma-Separated Values
DDoS	Distributed Denial-of-Service
DNS	Domain Name System
DRY	Don’t Repeat Yourself
EAI	Enterprise Application Integration
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
HR	Human Resources
HRM	Human Resource Management
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IAM	Identity and Access Management
ICT	Information and Communications Technology
iPaaS	Integration Platform as a Service
IT	Information Technology
JSON	JavaScript Object Notation
JWT	JSON Web Token
LeSS	Large Scale Scrum
M-form	Multidivisional Form
MB	Megabyte
MRP	Material Requirements Planning
NFS	Network File System
OAuth	Open Authorization
opex	Operating Expense
OS	Operating System
P2P	Point-to-Point

PaaS	Platform as a Service
REST	Representational State Transfer
RMI	Remote Method Invocation
RPI	Remote Procedure Invocation
RQ	Research Question
SaaS	Software as a Service
SAFe	Scaled Agile Framework
SCM	Supply Chain Management
SDK	Software Development Kit
SES	Simple Email Service
SFTP	Secure File Transfer Protocol
SNS	Simple Notification Service
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQS	Simple Queue Service
SRE	Site Reliability Engineering
SSM	AWS Systems Manager
TCT	Transaction Cost Theory
TLS	Transport Layer Security
U-form	Unitary Form
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1 Introduction

1.1 Background

Serverless is a recent trend in cloud computing that has increasingly been adopted in the IT industry. It is offered as a suite of managed services by all major public cloud providers including Amazon Web Services, Microsoft Azure and Google Cloud Platform, and has been described as the “next-generation cloud” due to its on-demand pricing and full application “as a service” functionality (McKinsey 2020). The pay-as-you-go pricing of serverless has allowed businesses to shift IT costs from a capex to opex model, in many cases dramatically reducing overall spending. Meanwhile, its full application functionality has enabled software organizations to greatly decrease their development complexity by eliminating the maintenance overhead associated with managing IT infrastructure, thereby freeing up developers to focus on developing value-added features.

Discussions around the benefits of serverless have generally focused on its potential for production cost savings, as that is usually perceived by buyers as the most salient criteria when deciding between containerized and serverless microservices in the cloud. In many cases, particularly when load is characterized by alternating peaks and periods of idleness, serverless emerges as a clear winner. However, there is yet little general guidance on how serverless should be extended more broadly into an organization’s enterprise architecture. This is especially the case in Enterprise Application Integration (EAI). Approaches to EAI have varied greatly ranging from distributed point-to-point integrations to centralized messaging-based Enterprise Service Bus (ESB) systems, and the rationales behind their designs are not always clear nor consistent. This makes it difficult to understand exactly what problems serverless architecture is meant to solve and, consequently, how to optimally implement it in EAI.

Personally, I first encountered serverless in EAI while working as a software engineer in the microservices team of a large multinational company. With several thousand employees and millions of customers globally, the company maintained a vast number of IT systems that were tightly integrated with its B2C digital channels for functions such as sales, customer service and marketing. Driven by the need to integrate these systems, the company had taken a commercial Enterprise Service Bus (ESB) product into use, which provided the centralized integration hub through which its end customers – on channels such as the ecommerce website and mobile application – could interface with core back-office information systems applications such as its CRM. The ESB was supposed to make things easier. Instead, it became one of the primary bottlenecks in the software development process. Data transformations needed to be coded and maintained in the

ESB's clunky desktop client and required using an abstruse programming language that was not widely known. The complexity of the ESB meant that dedicated personnel were needed to maintain it, and moreover these people were difficult to find as the skills required were not readily available in the job market. Furthermore, it became apparent that the ESB had become a common blocker in the development process for many of the integrating applications due to its tight coupling with them. Any change to data schemas in applications must also be updated in the schema transformations of the ESB, which effectively made the ESB tightly coupled with each of the integrating applications and therefore a single point of failure. After several years of operating the ESB, the company decided to migrate away from the ESB, and through an "exit project" moved all its integrations over to decentralized serverless architecture.

What was the driving force behind this series of transformations? Why did the ESB - a system designed to solve the problems of distributed application integration using centralization - ultimately end up being dismantled in favor of a decentralized, serverless approach? In the ensuing chapters, I will attempt to establish a theoretical framework for understanding how such widely divergent approaches to EAI emerge to make sense of cases like the one I described.

1.2 Problem Statement

The problem to be addressed in this thesis can be summarized by the following:

Serverless computing is increasingly being used as an architectural model in EAI, but there is at yet little consensus regarding how it should be implemented or evaluated.

Approaching this problem requires a normative framework for EAI - that is, a framework that can determine what constitutes a good or optimal EAI architecture - against which serverless can be evaluated. A review of the literature on EAI finds that no commonly agreed framework exists. Although there is a rich body of literature regarding integration patterns, such as those identified by Hohpe and Woolf (2003), it is not always clear *why* certain patterns are preferable to others. Indeed, this is the nature of design patterns - they formalize tried-and-tested approaches harvested from past solutions which together form a practical pattern language and are therefore fundamentally descriptive rather than normative. This makes it difficult to contextualize and evaluate radically new paradigms such as serverless computing. Because of this, I must first produce a holistic framework where different approaches to EAI can be understood and evaluated before I am able to properly contextualize the current use of serverless EAI architecture and evaluate its efficacy. A successful outcome of this research would be to provide an organization's

infrastructure or backend team a better understanding of the tradeoffs involved in adopting serverless architecture in EAI, as well as a rough blueprint for how to implement some integration patterns using serverless technologies.

These objectives can be formulated as the following three research questions, which I will attempt to answer throughout the thesis:

RQ1: What are the main existing approaches to EAI and what are their main problems?
RQ2: How can serverless be used in EAI?
RQ3: How well does serverless solve the main problems of EAI?

The main target audience of this research are software engineers, solutions architects, development managers and product owners of companies where serverless is being considered as part of the organization's cloud EAI architecture. The theoretical portions of this research focus on *what* EAI and serverless are, and *why* they have evolved into their current states. I explore the rationale behind different design patterns and the tradeoffs they entail, which I believe would be of value to IT decision-makers such as development managers and product owners. The creative portions of the thesis, namely the case design and implementation, focus on *how* serverless can be used for EAI, and are aimed at technical stakeholders such as software engineers and solutions architects who might be interested in seeing practical examples of how to implement a serverless integration runtime. Evaluating how well the design and implementation artifacts perform in turn allows us to further determine why serverless should or should not be used for an organization's EAI architecture, bringing us closer to having a prescriptive framework for decision-makers.

This research uses the design science research methodology, which seeks to acquire knowledge and achieve understanding of a problem domain and its solution by building and applying a designed artifact (Vaishnavi, Kuechler & Petter 2004; Hevner, March, Park & Ram 2004). A design theory is created through the process of developing and testing an information systems artifact, which in turn provides "a means to contribute knowledge by offering prescriptive statements and specification of outcomes for a system developed based on the theory".

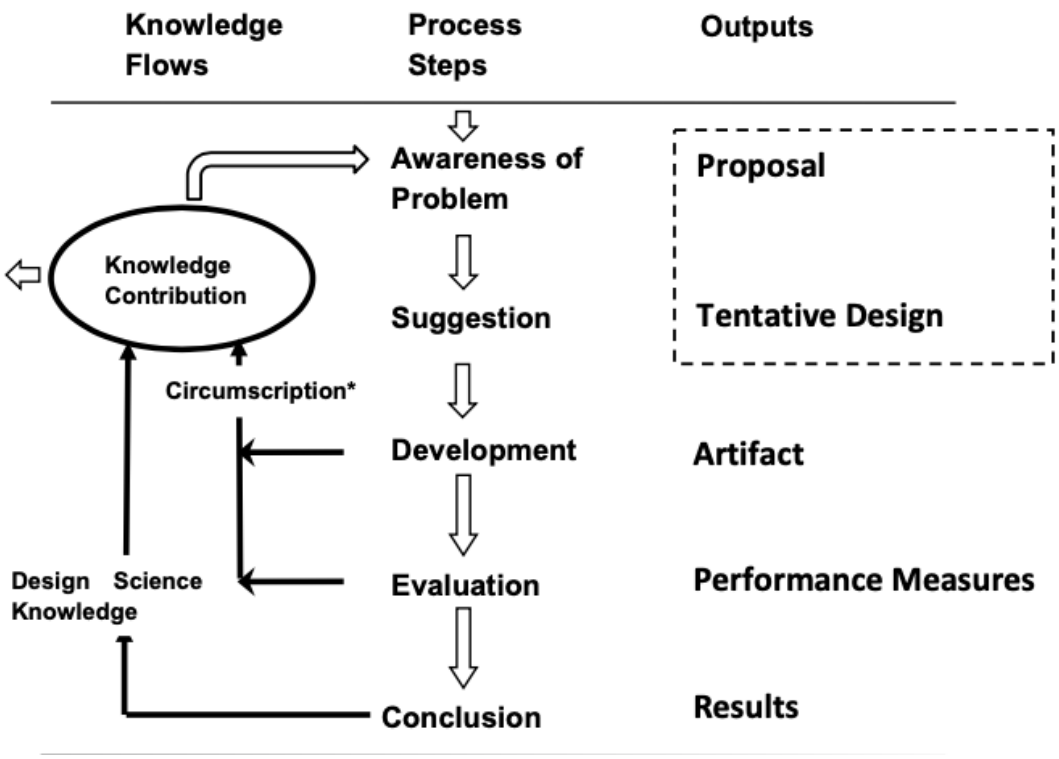

Figure 1. Design Science Research Process Model from Vaishnavi, Kuechler & Petter (2004)

A process model formulated by Vaishnavi, Kuechler & Petter (2004) summarizes the key process steps involved in the design science research methodology which will be undertaken in this research. The application of Vaishnavi's process model to this research can be summarized in the following steps:

(1) **Awareness of problem:** review the existing literature related to information systems and EAI architectures, describe existing approaches and highlight the main problems.
(2) **Suggestion:** develop a theoretical framework and propose an abstract model for evaluating the performance of current and potential solutions; present serverless computing as a solution to the main problems of EAI.
(3) **Development:** produce a design and instantiation of serverless integration architecture as an artifact (instantiation).
(4) **Evaluation:** evaluate the artifact using the abstract model developed in (1) and provide an assessment of its utility, quality, and efficacy.
(5) **Conclusion**: Summarize the knowledge that has been gained through this research and suggest possible future studies that can be undertaken.

The following activities and objectives are not included in the project: (i) Full source code of the case implementation - the focus of this research is on high-level design rather than low-level code details, so code snippets are used sparingly throughout the text and in the appendices; (ii) Real-world scenarios and data - this research is limited to a hypothetical scenario in order to focus on integration architecture rather than the integration details of

each application (e.g. data models, schemas, APIs), meaning that minimal “real” enterprise applications are used.

This thesis is structured as follows. In chapter two, I review the existing literature on EAI and provide a theoretical framework for evaluating integration approaches from the perspective of information systems, technology, organization, and economics. In chapter three, I provide a descriptive account of the main approaches that have been used in EAI and present serverless computing as the latest evolution in a series of improvements. In chapters four and five, I go through the design and implementation of a serverless integration architecture using Amazon Web Services (AWS). Finally, in chapter six, I evaluate the serverless architecture we developed in chapter four using the theoretical framework formulated from section one and conclude with some comments on further research

2 Theoretical Framework

2.1 Definition and Scope

Enterprise application integration (EAI) is the practice of enabling independently designed software to work together within an organization through the sharing of data, business rules and processes (Gartner 2021). Originally coined to describe integration practices for a growing number of business software applications in the 1990s, the term EAI has come to refer to both the architectural patterns used for managing software integrations within an organization, as well commercially available integration platform as a service (iPaaS) and enterprise service bus (ESB) software products that have developed precisely to implement these integrations such as IBM MQ, Oracle Service Bus, Microsoft Biztalk, MuleSoft, Informatica and Boomi. In this paper, we focus on the architectural definition of EAI which refers to the integration patterns between any information systems software used within an organization, from large-scale “enterprise” Software-as-a-Service (SaaS) applications to small in-house microservices.

The definition of EAI in this research is limited to system integration patterns (i) *within an organization*, and (ii) *between independently designed applications*. System integration is a broad topic in engineering that concerns the aggregation of and coordination between subsystems and encompasses everything from software applications to telecommunication networks. Our focus with EAI is on the subset of system integrations found between software applications within an organization - i.e., within an enterprise, corporation, business, or other social entity that has multiple members working together in pursuit of a common goal. Furthermore, the integrations that fall under our scope are those between *independently designed applications*, meaning integrations between applications that are not necessarily designed to work together. This contrasts with integrations between bespoke components of a software platform, for example, as in those cases components may be designed with the specific purpose of operating together as parts of a coherent system. A typical example scenario that falls under our research scope would be a company that integrates several third-party enterprise software applications such as HR, payroll, and financial accounting information systems.

2.2 Information Systems and Enterprise Applications

The rise of the term *enterprise software* dates to the 1990s, during which its usage began to spread rapidly. As noted in *TechCrunch*, the adoption of the term by the software industry actually owes to the television show *Star Trek: The Next Generation* - through a two-year licensing agreement, the software company Boole & Babbage (now BMC Software) launched a Star Trek-branded infomercial campaign in 1993 where it marketed

itself as the foremost “Enterprise Automation Company”, catalyzing other companies in the industry such as SAP, Baan and Lotus to embrace the “Enterprise” moniker in their product offerings (Aziz March 2020). In fact, prior to 1993, software applications that served the needs of organizations were much more likely to be referred to as *information systems software*. Every organization has data that needs to be collected, stored, processed, and distributed, and with the rise of computers, software applications have served as the main technological enabler for the management of these information systems.

The arrival of Enterprise Resource Planning (ERP) software to the manufacturing industry in the 1990’s proved to be the turning point that spawned the multitude of enterprise software applications that we see today. Whereas its precursors - Materials Requirements Planning (MRP I) and Manufacturing Resource Planning (MRP II) systems - were used to coordinate departments for manufacturing operations, procurement and production schedules, ERP enabled the full integration of all departments within a business across one single database. Thus, one can view ERP systems as an early attempt to achieve the goals of EAI through bundling critical business information systems into one software suite of integrated applications. Since the 1990s, information technology has grown into more and more business areas, with different departments such as finance, logistics, sales, marketing, and HR adopting purpose-built software applications to meet their needs. Driven by competitive pressures to lower development costs and shorten application life cycles, organizations have increasingly opted to use off-the-shelf software instead of having to reinvent the wheel, and this is reflected in the steady expansion of the enterprise software market over the past decade.

Today, the enterprise software market offers applications categorized by business functions such as accounting, business intelligence (BI), business process management (BPM), content management (CM), customer relationship management (CRM), human resource management (HRM) and supply chain management (SCM). Organizations often have no choice but to adopt off-the-shelf applications into their business information systems to stay competitive. While early enterprise applications were typically hosted on-premises within an organization’s IT infrastructure, the ubiquitous adoption of the internet has meant that they have increasingly been offered as cloud services accessible via the internet. Improvements in web technology and internet infrastructure during the past two decades have further made it possible to achieve highly advanced functionality within web applications, leading to increased adoption of web-based enterprise Software-as-a-Service (SaaS) offerings. For example, Salesforce.com, Workday and NetSuite - popular CRM, HR, and financial management applications respectively - are all web-based SaaS

platforms accessed primarily via HTTP through a web browser, mobile device, or web service APIs.

Enterprise software has acquired something of a bad name over the years for being big, slow, and ugly, and the phrase “enterprise software sucks” has become almost a rallying cry. This is not without reason. A study by Forrester Research commissioned by Sapho in 2016 found that despite significant growth in the use of enterprise systems and applications, 75% of employees had a hard time accessing information from them (Sapho 2016). Because enterprise applications typically form a critical component of an organization’s core processes, they often end up becoming “legacy” software that are difficult to maintain but nonetheless vital for supporting business operations. Perhaps because of this, recent years have seen a surge in innovative companies that promise greater speed and agility in their enterprise offerings by leveraging cloud computing, open-source components, and continuous delivery practices (Williams August 2020).

The reasons are myriad for why enterprise applications have become synonymous with big, slow, and legacy software, with some pointing to poor design, nonoptimal value streams in B2B sales and within organizations, and conservative corporate culture as a few culprits (Nygard 20 February 2009). However, I believe that there is a much more basic reason for the reported inefficiencies of enterprise software. Even when an enterprise application is fast, well-designed, and functional, it can be difficult to use simply because it is not well-integrated with a host of other applications within an organization’s network. The increasing scope and complexity of business information systems mean that integration between applications can easily become the primary bottleneck. To understand why “enterprise software sucks”, we need to look beyond their use in isolation, and to the broader information systems within which they operate.

2.3 What is Integration?

As the number of software applications grows within an organization, so does the need to facilitate communication and data sharing between them. Except for integrations provided through standard out-of-the-box adapters and connectors - for example, the marketing automation platform HubSpot’s standard connector to Salesforce (HubSpot 2021) - most applications operate in silos and cannot communicate with one another by default. End users generally access applications through an interface such as a desktop client, web browser, or mobile phone, and therefore the most primitive integration process one can conceive of would simply be for a person to periodically export and import data manually between two applications using these interfaces. Clearly, this is not a sustainable strategy - such manual processes would be costly, prone to error, and extremely limited in

scalability. The purpose of EAI is to achieve coordination between applications by enabling them to share data, business rules and processes with one another efficiently and securely through automation.

In a survey of over 380 experts from 44 countries on the topic of API integrations conducted by Cloud Elements, 83% of respondents reported that API integration is either critical or very important to their greater business strategy, with digital transformation (40.3%) and cloud app adoption (27.8%) being the primary reasons (Charboneau 15 June 2020). Most of the respondents (60.6%) indicated that their integration requirements are mostly cloud-based, suggesting that on-premises integrations are dwindling, and in terms of investments in integration infrastructure, API management is leading the way over iPaaS, SaaS, Enterprise Service Bus, and Message-Oriented Middleware. The highest area of demand for integration is for customized APIs that fit specific business processes (54.4% of respondents), reflecting the growing customization of digital services within organizations and the resulting need to build customized integrations.

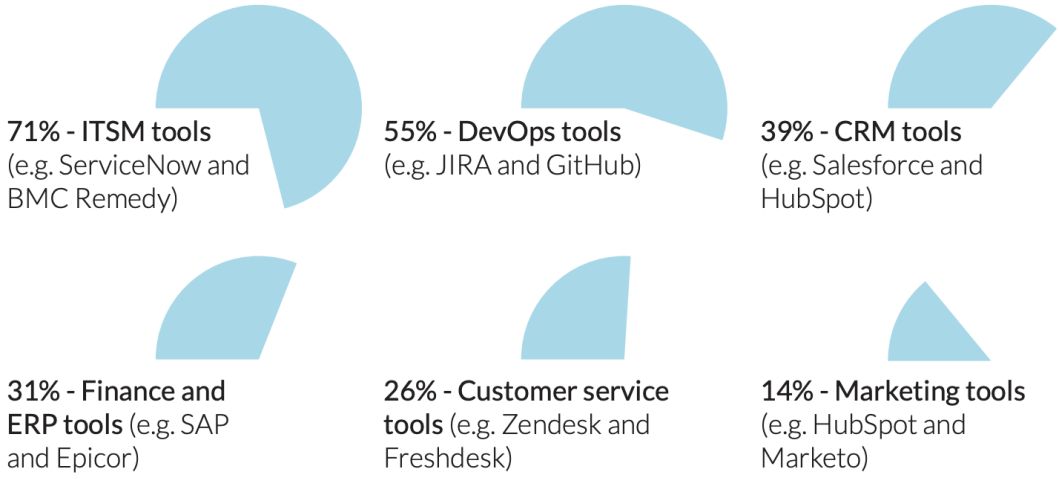

Figure 2. Distribution of types of applications in enterprise integration, from ONEiO's Integration 2020 report (ONEiO 2020).

A more granular insight into the environments and motives of application integration in organizations can be found in Finnish iPaaS company ONEiO's survey of 100 businesses on application integration. The survey found that IT Service Management (ITSM), DevOps and CRM tools are the most common types of applications in enterprise integration, followed by Finance and ERP, customer service and marketing tools. When asked what value integration creates for their business, greater levels of collaboration came out as the top response, followed by higher levels of customer service, faster approaches to change, becoming more agile and easier adoption of new technology.

Table 1. Responses to the question ‘What value do you see integration creating for your business?’ from ONEiO’s Integration 2020 Report (ONEiO 2020)

What value do you see integration creating for your business	Percentage of respondents
Greater levels of collaboration	71%
Provide higher levels of customer service	64%
Faster approaches to change	58%
To become more agile	57%
Easier adoption of new technology	47%
Gain better ROI on our software	37%
Launch products and features more frequently	29%
More competitive in the marketplace	23%

2.3.1 Hierarchies and Networks

Having described the purpose of integrations and examined some of their main use cases, we can now look at different approaches to designing integration architectures. Gulledge (2006) distinguishes two main categories of software integration which he terms “Big I” and “Little i”: While *Big I* integration can be described as “having all relevant data aligned with a single data model and stored only once”, *Little i* integration is the “the interfacing of systems together so they can pass information across a complex technology landscape”. The former, reminiscent of the large ERP systems of the 1990s, seeks to establish integration through centralization and vertical control over subcomponents, whereas the latter seeks to establish integration through the coordination of distributed interfaces over a network. This distinction, as we shall see, is at the heart of software design as well as enterprise application integration.

When designing software systems, a commonly encountered dilemma is the choice between whether to use a small number of large systems or a large number of small systems. The former “large systems” approach is what is sometimes referred to as a *monolith architecture*, while the latter “small systems” approach is sometimes referred to as a *distributed* or *microservices architecture*. The two approaches differ in their degree of centralization - monoliths are highly centralized, while microservices are distributed. Although this distinction is manifest in software engineering, it exists more broadly in all manner of information, social and natural systems. In fields as diverse as biology (Bechtel 2020), complex systems (Corominas-Murtra, Goñi, Sole & Rodríguez-Caso 2013), political science (Jung & Lake 2011), history (Ferguson 2017) and management theory (Kotter 2011) highly centralized systems have been modelled and referred to as *hierarchies* and

distributed systems as *networks*. Additionally, systems with clustered networks in which parties engage in exchange are typically referred to as *markets*, although these structures are relatively less prevalent in software and information systems.

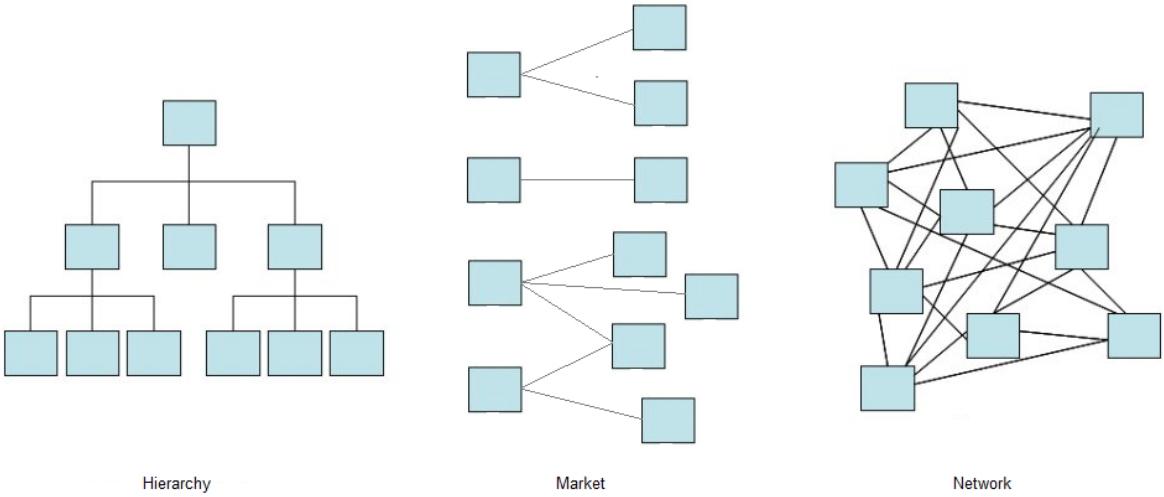

Figure 3. Hierarchy, market, and network models of social organization (Tubaro, 7 April 2016)

Borrowing from these fields, I will refer to the “large systems” approach (Gulledge’s *Big I* integration) as a *vertical hierarchy,* and the “small systems” approach (Gulledge’s *Little i* integration) as a *horizontal network*. As economic historian Ferguson explains in his exploration of the roles hierarchies and networks have played in economic and political contexts throughout history, while hierarchies are “vertical organizations characterized by centralized and top-down command, control, and communication”, networks are “spontaneously self-organizing, horizontal structures” requiring “minimal premeditation and leadership”.

Table 2. Comparison of high-level characteristics of hierarchy and network.

Hierarchy	**Network**
Vertical	Horizontal
Centralized	Decentralized, distributed
Top-down command, control and communication	Self-organizing

2.3.2 Technology and Organization

Applying the dichotomy between vertical hierarchies and horizontal networks to the field of software engineering allows us to understand the tradeoffs that lie at the intersection of technological and organizational factors, based on which we can evaluate the value of

different approaches to EAI. Technological factors are computational and generally deterministic processes affecting measures such as system design, application performance, computing costs, memory usage, energy consumption and algorithmic complexity - factors, in other words, which do not involve active, continual human agency. Organizational factors, on the other hand, are human, social, and behavioral processes such as development practices, communication, knowledge management, code maintainability, usability, and team efficiency, which are considered more subjective and are highly dependent on social cooperation. For example, a computer program may be technologically optimal by having low-order algorithmic complexity, efficient memory usage and low computing cost, but at the same time organizationally nonoptimal if it is difficult to understand and maintain. Having a holistic framework that takes both technological and organizational factors into account is thus necessary to gain a complete understanding of the divergent approaches in systems architecture and design. Some examples of organizational factors are highlighted in Hevner's Information Systems Research framework (2004) which identifies personal roles, responsibilities, and characteristics as well as organizational strategies, structure, culture, and processes as relevant environmental factors in the formation of business needs in information systems research.

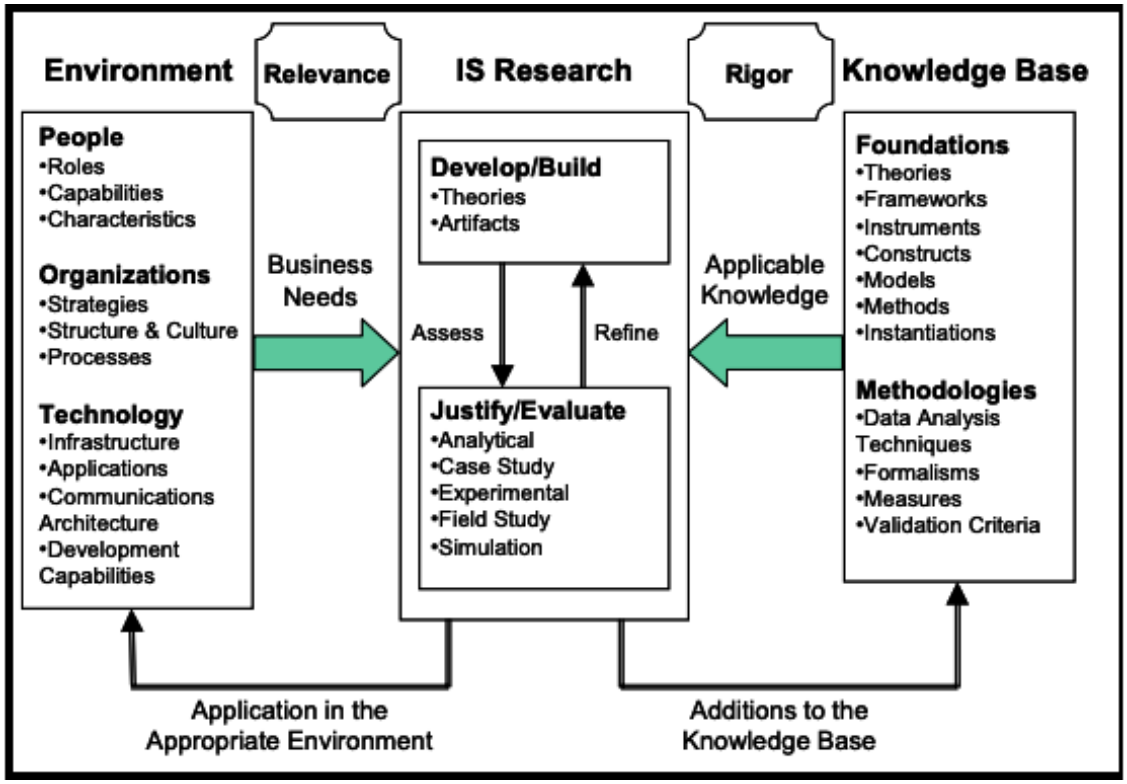

Figure 4. Information Systems Research framework as formulated by Hevner (Hevner & al. 2004, 80)

With these factors in mind, we can see that divergent approaches to many software design problems can be explained in terms of tradeoffs between the costs incurred by

hierarchical and network-based organizational structures. Proponents of vertical hierarchy advocate for principles such as “don’t repeat yourself” (or DRY) and data normalization, which tend to result in monolithic structures such as shared databases, tight coupling between subcomponents and high levels of code abstraction. These choices are based on the rationale that they bring about greater data consistency, improved enforcement of contracts between software components resulting in fewer human errors, lower technological costs (e.g., less idle computing capacity) from economies of scale and efficiency from reusable components. The DRY principle states (Hunt & Thomas, 2000):

> “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

In practice, this means avoiding unnecessary duplication and its resultant complexity by using abstractions and data normalization. In software architecture, the DRY principle can be found in the object-oriented programming mechanism of class inheritance: a base class is used as the blueprint from which sub-classes are derived. In database design, it is found in the use of database normalization where a database is structured according to a series of so-called normal forms in order to reduce data redundancy and improve data integrity. The DRY principle is therefore emblematic of hierarchical design – complexity is reduced by imposing a single, authoritative source of control to which subsystems are subordinate.

On the other hand, proponents of horizontal networks actively resist normalization and code abstraction in favor of loose coupling, independent deployability, and encapsulation, which tend towards patterns such as agile and microservices architecture. These choices are based on the idea that they enable teams to develop applications more autonomously, thereby reducing costs incurred from interdependency risks, increasing experimentation, decreasing cycle time, and improving continuous delivery.

In EAI, hierarchies and networks similarly underlie divergent approaches to structuring coordination between applications. An enterprise consisting of independently designed applications is, by definition, a decentralized network. However, integration architectures diverge in the degree of hierarchy which they impose on the network, and just as there are differences in software design between monoliths and microservices, there are differences in integration design regarding just how hierarchical or decentralized an integration architecture should be. An evaluative framework would therefore be valuable to decision-makers who need to choose between implementing either a vertical hierarchy or a horizontal network in their integration architecture. Such a framework would need to be able to evaluate both technological and organizational factors, which work in tandem to

produce value in an organization. In fact, when analyzing system architecture as we are doing in this paper, we must find a level of abstraction where human agents (e.g., programmers, software users) and computer systems (e.g., software applications, integration systems) can be modelled transitively based on a set of shared, fundamental organizational structures.

Borrowing from Malone and Smith (Malone & Smith 1984), we establish this level of abstraction by defining an *organization* as a group of *agents* which consist of either people or machines. To *organize* is to establish the goals of the organization, segment the goals into separate activities, and assign the activities to agents in such a way that the overall goals are achieved. This abstraction allows us to evaluate technological and organizational factors as interchangeable parts of shared processes. In the context of EAI, the assignment of activities to agents - referred to by Malone and Smith as the “task assignment problem” - is the orchestration of integrations between applications, which consists of both organizational processes as well as specific integration technologies.

2.4 Normative Frameworks

Having examined the role technological and organizational factors play in EAI, we can now begin exploring what constitutes a good EAI architecture by developing a theoretical framework with which we can evaluate different approaches. We do so by drawing from several normative frameworks that have been used in the context of information systems: information system agility, efficiency and scalability, and transaction cost economics.

2.4.1 Information System Agility

One approach to evaluating EAI is to refer to descriptive characteristics that are considered as norms. In recent years, agility has increasingly been regarded as a norm, being embraced by organizations from small startups to Fortune-500 companies. Large corporations have adopted Scaled Agile Framework (SAFe) and Large-Scale Scrum (LeSS) in their organizations, and a growing number of agile experts have created something of a cottage industry providing coaching, training, and certification in agile methodologies. Given its prevalence, can information system agility be used as the normative framework - the touchstone or measuring stick, so to speak - against which the technological and organizational factors of EAI architectures can be evaluated?

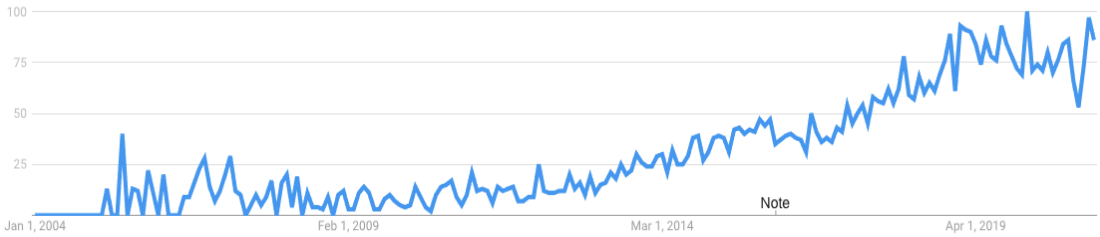

Figure 5. Google Trends data for worldwide interest in topic “Agile Framework” since 2004

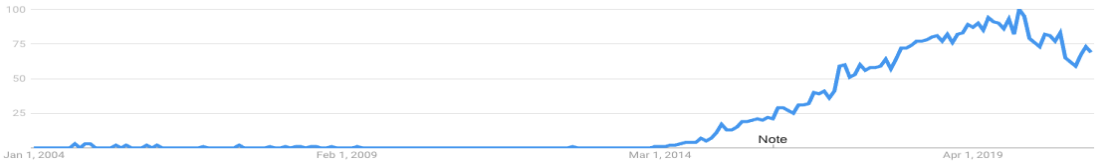

Figure 6. Google Trends data for worldwide interest in the topic “Microservices” since 2004

Chaudhary, Hyde and Rodger (2017) define information system agility as the ability of an information system to “sense a change in real time; diagnose it in real time; and select and execute a response in real time”. A list of benefits compiled from academic literature and validated using qualitative empirical data from practitioners include reduced time to implement changes, increased robustness of implemented changes, increased efficiency and effectiveness of existing business processes, and the ability to respond rapidly to security threats like virus attacks and cyberattacks. Taking a hermeneutic approach to analyzing information systems, it seems that a good EAI architecture is one that increases information system agility.

How, then, can EAI architecture increase information system agility? One answer lies in organizational communication structure and its relationship with system design. The organizational factors described by information system agility are largely related to communication, such as team structure, interdependency management, task prioritization and so on. In the software industry, it has been widely observed that the design structure of an organization’s software tends to mirror its communication structure. This mirroring between systems design and organizational structure is codified by Conway’s law (Conway 1968), which states:

> “Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.”

Conway's law has been supported by empirical research, most notably a study of the "mirroring hypothesis" comparing open-source software organizations to commercial ones which found that loosely coupled organizations in the software industry tend to produce more modular designs (MacCormack, Baldwin & Rusnak 2011). While Conway's law hints at causality flowing from organizational structure to systems design, we can also infer causality as flowing in the opposite direction. To take MacCormack's case of open-source software organizations, for example, one can point to distributed version control systems as the key technology that enables loosely coupled organizational structure.

Being the centerpiece responsible for enabling coordination between different systems, EAI serves as the bedrock from which an organization's communication structure emerges. Using the characteristics of vertical hierarchies and horizontal networks, we can compare software design structures to agile organizations to identify what types of EAI design are most conducive to agility. Agile characteristics such as "collaboration" and "self-organizing teams" match quite well our definition of horizontal networks as "spontaneously self-organizing, horizontal structures" requiring "minimal premeditation and leadership". Indeed, technologies that we have described as being based on horizontal networks, such as microservices, are often implemented in the context of agile transformation.

In other words, for an organization to achieve information system agility, it needs to have an EAI architecture in place that enables coordination between applications to be self-organized, independent, and loosely coupled. This fits conveniently into the current technological trend of microservices who's decentralized and network-like characteristics mirror those of the agile organizations in which they are used.

Table 3. Comparison of hierarchies and networks in the context of technological factors

Technology	**Hierarchy**	**Network**
Infrastructure	Shared	Independent
Applications	Monolith	Microservice components
Communications Architecture	Closely coupled Hub-and-spoke	Loosely coupled Clustered, point-to-point
Development capabilities	Code reuse Shared database Tight coupling	Duplicated code and data Database-per-service Loose coupling

Table 4. Comparison of hierarchies and networks in the context of organizational factors

Organizations	**Hierarchy**	**Network**
Strategies	Integration management Standardization Economies of scale	Self-organization Agility Experimentation
Structure & Culture	Function-based Tightly-coupled	Product-based Loosely-coupled
Processes	Phase-gate Push-based	Scrum / Kanban Pull-based

Table 5. Comparison of hierarchies and networks in the context of people factors

People	**Hierarchy**	**Network**
Roles	Project Manager Development Manager	Product Owner Scrum Master
Capabilities	Specialized	End-to-end (full stack)
Characteristics	Top-down decision-making	Bottom-up decision-making

However, I find this approach to be incomplete and rather limiting as an evaluative framework. Information system agility simply highlights that characteristics of horizontal networks and decentralization are valued in certain aspects of organizational structure - it is not enough to conclude that we should therefore always design systems with characteristics of a horizontal network. In fact, as we shall later discuss, an organization is fundamentally a hierarchical structure, and agility is a way to gain competitive advantage through selective network characteristics.

This is true also for EAI - integration is fundamentally a hierarchical process imposed upon a network but incorporating network characteristics such as loose coupling and self-organization may increase its efficiency. The rising popularity of decentralized, network-based technologies certainly does not invalidate the benefits of hierarchical ones, as we can see in the case of monolithic applications and more recently with the resurgence of the monorepo version control strategy. Hierarchical patterns are in many cases still preferred by organizations over decentralized design for a multitude of reasons.

The real question, therefore, is why selective network characteristics are considered more efficient within fundamentally hierarchical organizations (for example, self-organizing agile software teams within a company that has management and a CEO). To have a more robust normative framework, we need to examine more fundamental normative criteria than the ones offered by agility. So, it is to economic efficiency and cost benefit analysis we turn to next.

2.4.2 Efficiency and Flexibility

The most basic normative criteria shared by all industrial organizations is economic efficiency, with cost-benefit analysis (CBA) being the primary tool for evaluation. In the domain of information systems and software engineering, this is not as straightforward as calculating the production cost savings of a technology under evaluation. As our earlier analysis has shown, technological choices can have large spillover effects on organizational structure, meaning that organizational impact needs to be factored into the CBA of any technology choice. This can be challenging, as the benefits of organizational factors such as collaboration are difficult to quantify and therefore easily hidden.

Malone (1985) provides a model of organizational structure defined in terms of tradeoffs between *production costs*, *coordination costs*, and *vulnerability costs*. Production costs are the “costs of performing the basic tasks necessary to achieve the organization's goals” (Malone 1985, 10), which in the context of information systems include the costs of computing infrastructure, software licenses, wages, etc. Coordination costs are the “overhead” required to manage and delegate tasks within an organization, which typically fall under the purview of managers. Vulnerability costs are the “unavoidable costs of a changed situation that are incurred before the organization can adapt to the new situation” (Malone 1985, 10) - for example, lost competitiveness during economic or technology shock. These costs underlie organizational *efficiency*, consisting of production and coordination costs, and *flexibility*, consisting of coordination and vulnerability costs.

Table 6. Examples of U-Form and M-form governance structures from Qian, Roland & Xu (1999)

	U-form	M-form
Government	France and Japan: Power centralized in specialized ministries. Regional governments have few powers.	American federalism: large powers held by states. States initiate and experiment with changes. Successful initiatives are imitated by other states.
Economy	Soviet “branch organization”: production is organized into functional ministries (mining, machinery, textile, etc.) each responsible for a specialization and controlling gigantic factories.	Chinese “regional organization”: production organized geographically (provinces, prefectures, counties, townships, villages) into semi-autonomous units.
Corporation	Ford Motor Company (pre-WWII): highly specialized departments (production, sales, purchasing) each dependent on one another	General Motors under Alfred Sloan: collection of self-contained divisions (Chevrolet, Pontiac, Oldsmobile)

In this model, we can define two main organizational forms in terms of the tradeoffs they make between production, coordination, and vulnerability costs: these are the *U-form* (unitary form) and *M-form* (multidivisional form) organization (Qian, Roland & Xu 1999). A U-form organization groups similar tasks into functional units which are shared among products, where operational processing and decision making of all products must be coordinated through a central “executive office”. This organizational form achieves efficiency through specialization and economies of scale, thereby economizing production costs at the expense of incurring relatively high coordination costs.

An M-form organization, on the other hand, is decomposed into self-contained units containing complementary tasks which require relatively little coordination with a centralized hub. It therefore achieves efficiency and flexibility by economizing coordination costs, at the expense of incurring relatively high production costs from lack of functional specialization or economies of scale. While Malone suggests that neither organizational form is better equipped than the other to mitigate vulnerability costs, Qian, Roland, and Xu (2006) provide evidence to the contrary. The flexibility resulting from coordination advantages of M-form organizations, Qian & et al. argue, effectively lowers the cost of learning and results in higher propensity to innovate, which confers a dynamic advantage allowing M-form organizations to mitigate vulnerability costs more effectively.

A *market*, meanwhile, is an organizational form that falls outside of firm boundaries and which can be centralized, as in the case of the stock market (which is quite like a

functional hierarchy), or decentralized, as in a consumer market for automobiles. From the perspective of a firm, a market presents an organizational form where tasks can be completed outside of the firm boundaries, for example in cases where a task is outsourced to another firm. This relationship between firms and markets is at the heart of outsourcing and software-as-a-service and will be covered in more detail later when we discuss transaction cost economics. For now, it suffices to say that markets offer another option for task completion beyond U-form and M-form structures. In Malone's words (1985, 13):

> "One of the important insights from the literature of organizational theory and economics (e.g., see Williamson, 1975) is that the same tasks can, in principle, be coordinated by either a market or a hierarchy. For example, General Motors does not need to make all the components that go into its finished products. Instead of manufacturing its own tires, for instance, it can purchase tires from other suppliers. When it does this, it is using a market to coordinate the same activities (i.e., tire production) that would otherwise have been coordinated by hierarchical management structures within General Motors."

Based on Malone's model, information system agility can be understood as an organizational structure that optimizes flexibility. An agile organization mitigates the vulnerability costs of technology shock through its ability to "sense a change in real time; diagnose it in real time; and select and execute a response in real time", while also economizing on coordination costs through its ability to self-organize. In other words, we can interpret an agile organization as following an M-form governance structure - it achieves efficiency and flexibility by economizing on coordination costs, while forgoing economies of scale and incurring relatively higher production costs due to a lack of functional specialization.

Table 7. Comparison of U-form and M-form organizations in terms of production, coordination, and vulnerability costs

	U-form	M-form
Production costs	Low - specialization into reusable functions, economies of scale	High - replication due to self-contained units
Coordination costs	High - decision-making must be coordinated through a central "executive office"	Low - decision-making done by self-contained units
Vulnerability costs	High - lower propensity to innovate and adapt to shock	Low - higher propensity to innovate and adapt to shock

Returning to our distinction between vertical hierarchies and horizontal networks in EAI architecture, we see that the U-form organization aptly describes a vertical hierarchy and shares many characteristics with monolithic design and waterfall organizations. M-form organizations and markets, on the other hand, are self-organizing and comparatively decentralized, and thus have characteristics of a horizontal network. It is important to note here that in the terminology of economics, both U-form and M-form organizations are technically hierarchies, and therefore I run the risk of confusing the reader by calling M-Form organizations networks. To clarify my comparison, when I refer to the M-form organization as a network, it is primarily to contrast with the relatively hierarchical structure of the U-form organization. Drawing upon examples from Qian et al.'s empirical research and applying our insights from the previous section on Conway's Law, we can therefore categorize software and integration architectures under U-form and M-form organizations as illustrated in the table.

Table 8. Comparison of U-form and M-form organizational structures expressed in software teams, software architecture, integration architecture

	U-form (Hierarchy)	M-form (Network)
Software Teams	Waterfall: large, specialized teams (analysts, designers, developers, testers) each responsible for a certain stage in linear development. Centralized decision-making and integration management.	Agile: small, self-sufficient, and cross-functional teams (analysts, designers, developers, testers) with ownership of a product
Software Architecture	Monolithic application: single-tiered structure containing interdependent components; service-oriented architecture	Microservices: application composed of independent, distributed components each with a single responsibility
Integration Architecture	Centralized Hub-and-spoke Enterprise Service Bus	Distributed Microservices Serverless

Viewing hierarchical architectures as U-form and horizontal network architectures as M-form helps us understand the strategies that each one follows in terms of the tradeoffs they make. Hierarchical integration architectures and monolithic applications (i.e., U-form structures) have low production costs due to their advantage of being able to reuse code, share business processes and form specialized teams responsible for specific functionalities. This economizing on production costs is most evident in service-oriented architecture, where application components are encapsulated and shared as reusable services over an exposure gateway. By contrast, microservice and distributed system architectures (i.e., M-form structures) often require setting up a larger number of

computing environments, deploying replicated code, and having less specialized teams, thereby incurring greater production costs.

On the other hand, horizontal network architectures have lower coordination costs when compared to hierarchical architectures. In hierarchies such as a monolithic application, tightly coupled functional dependencies mean that a change in one component will likely have an impact on another component. Practically, this means that hierarchically structured software development teams frequently bottleneck each other in the development cycle, resulting in slower delivery and delayed time-to-market, as well as higher chances of incurring costs from unmitigated interdependency risks such as system failures. As briefly examined in the discussion concerning agility, hierarchical architectures also suffer from higher vulnerability costs. While microservice architectures enable teams to decide their own technologies, programming languages and tools, hierarchical architectures such as iPaaS or ESBs often require a standardized toolset and programming language. This is good for developing specialization and achieving efficiency through economies of scale, but if the standard toolset and programming language become supplanted by vastly superior technologies, the organization can suffer huge costs from lost competitiveness. Security vulnerabilities present a similar risk. Microservices, on the other hand, not only are able to adapt to shock much more quickly due to their low coordination costs but are also in a much better position to continually innovate and capitalize on disruptive technologies due to their flexibility from self-organization. In the information technology industry in particular, vulnerability costs can be existential threats, while the opposite - the ability to innovate through disruptive technologies - has often served as a ticket for market entry and, in some cases, dominance.

Using this model, we can evaluate the efficacy of an EAI architecture for a particular organization by analyzing its impacts on the organization’s production, coordination, and vulnerability costs. For example, an organization that has a highly effective communication structure with only a few employees, such as a small startup company, might not benefit much from economizing coordination costs, and therefore a hierarchical integration architecture that economizes on production costs may be the optimal choice. By contrast, a large company with many products and information systems might find that high coordination costs far outweigh production costs, in which case a vertical network EAI architecture may be the optimal choice. This not only provides us with a normative framework based on CBA which we can use to evaluate the efficacy of an EAI architecture, but also helps us understand how EAI has evolved through the years.

In the three decades since Malone formulated this model, however, several technological developments including the internet, personal computing, and vastly improved computational processing power have tipped the scale over to horizontal networks. These technological developments have dramatically lowered both production and coordination costs, which I believe partially accounts for the growing enthusiasm we see today for horizontal network strategies such as agile, microservices, cloud computing and serverless. Incredibly, Malone predicts some of these developments in 1985. First, Malone postulates that if the “unit costs” of coordination - i.e., transmitting and processing information - decrease, then “coordination mechanisms that would previously have been prohibitively expensive will, in some situations, become affordable.” (Malone 1985, 22) I believe this is precisely what has occurred as today’s instant messaging, project management, email, and version control technologies have rendered Malone’s concept of coordination costs almost unrecognizable. Many of today’s firms have low explicit coordination costs but extremely high interdependencies among units, a model that doesn’t quite fit into either U-form or M-form. Second, Malone predicts that information technology should lower transaction costs (a concept we will return to shortly) which should consequently make “markets more efficient and therefore more desirable as coordination mechanisms” (Malone 1985, 23). This is an incredibly prescient statement, as we have seen the meteoric rise of IT outsourcing, Software-as-a-Service, and cloud computing in the past two decades. Third, Malone predicts that the effect of decreasing coordination costs from IT will result in an increase in market-like organizational mechanisms within firms. In Malone’s words (Malone 1985, 24):

> “Another, and perhaps more likely, possibility is that coordination mechanisms like those in a market will come to be used more and more inside large firms. For example, the widespread use of electronic mail, computer conferencing, and electronic markets (e.g., Hiltz & Turoff, 1978; Johansen, 1984: Turoff, 1984) can facilitate what some observers (e.g., Mintzberg, 1979; Toffler, 1970) have called "adhocracies," that is, rapidly changing organizations with many shifting project teams composed of people with different skills and knowledge. These organizations can rely heavily on networks of lateral relations at all levels of the organization rather than relying solely on the hierarchical relations of traditional bureaucracies to coordinate people's work (e.g., Rogers, 1984; Naisbitt, 1983).”

Quite astoundingly, Malone’s description of these hypothetical “adhocracies” seems to exactly describe the agile teams we see today. As the drastic technological transformations postulated by Malone have been rendered a reality over the past three decades, they also bring with them complexities that are not accounted for in Malone’s original model. In particular, the market-like mechanisms that Malone postulates would emerge within large firms is now readily observable in the case of enterprise integrations and have in fact created novel difficulties that perhaps were not imaginable in 1985. To

understand what these difficulties are and how we can tackle them, we turn next to transaction cost economics, a field within economics that studies the complex relationship between firms and markets.

2.4.3 Transaction Cost Economics

Transaction cost economics has been used in information systems research as a theoretical framework for evaluating the economic impact of information technology (Ciborra 1983). It posits that “the optimum organizational structure is one that achieves economic efficiency by minimizing the costs of exchange” (Young 2013), where these costs of exchange can be understood broadly as the costs of running the economic system of firms. A subfield within the discipline of economics, transaction cost economics has primarily been applied to the analysis of firms and markets. Particularly, with its early formulation by Ronald Coase (1937), transaction cost economics examines why firms emerge in the first place when individuals can freely engage in bilateral trade through the price mechanism of markets. As Coase examines, “in view of the fact that it is usually argued that co-ordination will be done by the price mechanism, why is such organization necessary?”

The answer, as postulated by Coase and elaborated by Williamson (1991), is that firms emerge because they economize on the transaction costs of exchange and production more efficiently than individuals do. In markets, transaction costs are defined as the total costs of making a transaction (excluding, from a buyer’s perspective, the price of the good or service itself) and can be divided into three broad categories which constitute the different phases of a transaction:

1. *Search and information costs*, involved in determining the price and availability of a product or service on the market.
2. *Bargaining and decision costs*, such as contract negotiation, bidding, auctioning, and other activities required for two parties to come to an agreement on a transaction.
3. *Policing and enforcement costs*, namely the costs of monitoring and enforcing the terms of a transaction, including recourse to litigation when terms are violated.

Williamson, who received a Nobel Prize in Economics largely based on his work on transaction cost economics, theorized that where transaction costs are high, hierarchical governance structures such as firms will tend to form. On the other hand, where transaction costs are low, transactions will tend to take place between individuals in market structures, with minimal intervention from firms. Williamson thus provides us with a framework for understanding not only why firms emerge, but how they determine which transactions to include within their scope. As Coase explains when examining the boundaries of a firm’s scope (Coase 1937):

> “At the margin, the costs of organizing within the firm will be equal either to the costs of organizing in another firm or to the costs involved in leaving the transaction to be ‘organized’ by the price mechanism”.

In other words, transaction costs set the natural boundaries of a firm. As Malone accurately predicts, information technology has dramatically reduced transaction costs and consequently shifted firm boundaries and organizational mechanisms towards markets. We observe this readily in the growth of IT consulting and the SaaS, PaaS and IaaS industries. Information technology has made it incredibly easy for organizations to purchase software and services from a market rather than developing in-house competences and solutions, with cloud computing being a salient example. Instead of organizing IT infrastructure tasks through internal M-form or U-form structures, an organization can now resort to a market solution where a server can be provisioned in a matter of a few clicks and automatically billed electronically based on usage. From the perspective of the vendor, cloud computing can perhaps be analyzed as the product of a hierarchical U-form structure economizing on production costs through economies of scale - infrastructure-as-a service is born out of the cloud vendor’s extreme specialization in developing computing infrastructure and operations, which low transaction costs (i.e., internet) have enabled the commercialization of. From the perspective of a firm, however, cloud computing in fact serves as an enabler for a decentralized M-form organizational structure. With infrastructure tasks outsourced away, software units can be more self-organized and have less interdependencies with one another. This is perhaps what Malone means when he suggests that “coordination mechanisms like those in a market will come to be used more and more inside large firms” (Malone 2003, 24) - decentralized, on-demand services have become so highly integrated into enterprise architectures that the distinction between market and firm can become blurred, making organizations more and more characteristic of horizontal networks.

While transaction cost theory (TCT) has been used to explain the emergence and role of IT systems such as enterprise applications and electronic marketplaces in business (Laudon, Laudon & Brabston 2005), little has been written about its impact on information systems architecture. As pointed out by Cordella (2006), much of the information systems literature invoking TCT has presented IT systems as hierarchical structures which solve “inefficiencies in the organization of transactions in complex and uncertain settings” and focuses incorrectly on the direct positive effects such systems have on “information flow, distribution, and management”. For instance, an organization can dramatically decrease the transaction costs of finding and purchasing its products by setting up an online ecommerce store. But what about the costs incurred from online customer service,

handling online return policies, and maintaining the internet payment system? These are transaction costs that emerge because of the ecommerce system, and it would be short-sighted for an organization to not take them into account when making a technology decision. As Cordella explains, transaction costs "in fact often increase as a consequence of the adoption of ICT…because of the associated extra costs required to accommodate the more complex environment that emerges". Cordella's central point is that while studies in information systems research have invoked TCT to positively evaluate the adoption of ICT systems, they have largely failed to consider the negative externalities that these systems produce and are therefore flawed. Cordella does not argue that TCT is of no value to information systems research, only that the existence of these negative externalities in organizational environments forces us to rethink the way TCT should be used in evaluating the impacts of ICT systems.

Notwithstanding Cordella's critique of the use of TCT in information systems, we restate the function he presents for capturing the transaction costs of a specific exchange here (Cordella, 2006):

$$Tc = f(U; C; Br; Ia; As; Ob; Cc)$$

where Tc is transaction costs, U is uncertainty, C is complexity, Br is bounded rationality, Ia is information asymmetry, As is asset specificity, Ob is opportunistic behaviour, and Cc is coordination costs.

As the reader may recall from Malone's discussion, one of the impacts of IT on organizational form is that "coordination mechanisms like those in a market will come to be used more and more inside large firms" (Malone 2003, 24). Indeed, as we look at today's enterprise architectures, the boundaries between in-house and SaaS applications are blurring as we see an increasing number of software products being offered for all manner of business use cases. Effectively, the IT organizations of many large companies have moved away from developing in-house software to essentially becoming API brokers, tasked with integrating different commercial products or open-source software packages together. This phenomenon emerged because technologies such as the internet have enabled transaction costs to become so low that markets are more and more frequently able to outcompete and therefore displace organizational functions. The result is that the organizational environments that software teams find themselves in today might no longer bear resemblance to the neat U-form and M-form structures that Malone describes, but something more akin to M-form - i.e., decentralized, self-organizing agile units - with extremely high interdependencies between units as well as with units outside of firm boundaries (e.g., external SaaS integrations).

Given this background, we can model enterprise integrations through the transaction cost lens of market mechanisms. Assuming a hypothetical scenario in which an organization's enterprise architecture is governed fully by market mechanisms, let's take a *transaction* to refer to the implementation or maintenance of an integration between two applications. Extrapolating each of these factors from markets to EAI yields the following

Table 9. Transaction cost factors applied to the context of EAI

Transaction Cost Factor	**Application in the context of EAI architecture**
Uncertainty	Likelihood that the integration fails to be completed on time or fails to operate in production
Bounded rationality	Extent to which developers and other ICT stakeholders do not act rationally - they may make incorrect assumptions about technologies, misunderstand requirements and produce bugs, particularly as complexity increases
Information asymmetry	Extent to which individuals responsible for building and maintaining the integration do not have information about the data, business rules and processes involved in the integrating applications
Asset specificity	Degree to which the personnel, tools, time and other resources invested in developing the integration are specific to the integration and not redeployable in other contexts
Opportunistic behavior	Likelihood that stakeholders act in their own self-interest rather than the interest of completing the integration. Admittedly, this is a feature that is quite specific to the context of markets and is therefore not really relevant for intra-organizational analysis, since organizations are formed precisely to mitigate these opportunistic market behaviors. We will therefore not use this factor in our analysis. However, it could in theory be applied to analyze the incentives and behavior of development teams, particularly in cases where subcontracting or consulting is used, as those cases are indeed characterized by market transactions
Coordination costs	Costs of coordinating requirements and tasks between different teams, tools, and stakeholders (known as *integration management* in traditional project management)

Cordella points out that each of these factors have high interdependencies with one another, and therefore when evaluating their role in the design of an EAI architecture we are unlikely to be able isolate the effects of any single factor. As he explains, while previous information systems research has often attempted to reduce the transaction cost effects of a single factor, “Transactions costs are not only the sum of the costs generated by the different factors, but are influenced by the imbricate, interdependent relationship between them”. It would thus be mistaken to provide any prescriptive framework based on an isolated transaction cost factor - rather, these factors must be considered, in a sense, as an emergent whole.

Applying these transaction cost factors to the three different phases of a transaction by yields the following:

- *Search and information costs*: In EAI, this corresponds to interface discovery - namely, the costs involved in documenting, managing, and searching for integration interfaces such as API endpoints and specifications. As an example, a vertical hierarchy may lower the cost of finding an integration interface by centralizing them all into one exposure gateway (decreased information asymmetry). Depending on how this is implemented, however, this centralized exposure gateway may result in increased transaction costs caused by information overload (increased bounded rationality*)*, single-point-of-failure (increased uncertainty) and costs from specialized tooling (increased asset specificity)
- *Bargaining and decision costs*: This corresponds to the development required for protocol translation and data transformation between applications in EAI. Again, a vertical hierarchy may prevent the need to build separate transformations between each node by providing a hub-and-spoke architecture with a standard protocol and data model (decreased coordination costs), while at the same time incur greater sunk costs by investing in tools and personnel specific to the integration platform (increased asset specificity), and making deployment processes less transparent to those not involved in managing the integration hub (increased information asymmetry)
- *Policing and enforcement costs*: In EAI, this corresponds to the operations, administration, infrastructure management, monitoring and security analysis required to maintain the integration runtime.

Examining these transaction cost factors in the context of EAI architectures allows us to properly consider the emergent, negative externalities that EAI can produce in enterprise information systems, which in turn allows us to evaluate the suitability of an EAI architecture more accurately for an organization.

2.5 Abstract Model

Having reviewed the normative frameworks presented by information system agility, organizational efficiency and flexibility, and transaction cost economics, we now have the theoretical basis on which to develop an abstract model for evaluating different approaches to EAI. From our analysis of information system agility, we saw that agility can be defined more robustly in terms of efficiency and flexibility, which in turn can be expressed as tradeoffs between production, coordination, and vulnerability costs. These tradeoffs are identified in two main organizational forms, one defined by a vertical hierarchical (U-form) which economizes on production costs to achieve efficiency, and the other by a horizontal network (M-form) which economizes on coordination and vulnerability costs to achieve flexibility. Further, we found that as information technology dramatically lowers both the unit costs of coordination and the transaction costs of exchange, coordination mechanisms that were once prohibitively expensive - such as IT outsourcing, SaaS solutions and cloud computing - have become affordable, meaning that market mechanisms have effectively become an integral part of the organizational structure of firms. Therefore, the transaction costs of market mechanisms, particularly the costs of search and information, bargaining and decision, and policing and enforcement, must now be accounted for as negative externalities when evaluating the impact of an EAI architecture.

We can summarize this abstract evaluative model as follows:

An effective EAI architecture is one that economizes on the production, coordination, and vulnerability costs of organization, while minimizing negative externalities incurred from search, decision-making, and enforcement.

With this abstract model, we are now equipped with the concepts and language needed to describe and evaluate the different types of integration patterns and integration architectures that can be found in EAI.

3 Enterprise Application Integration

3.1 Integration Patterns

In *Enterprise Integration Patterns*, Hohpe and Woolf (2003) provide a pattern language for enterprise application integration that has been widely used in the software industry and implemented in EAI and message-oriented middleware frameworks such as Spring Integration and Apache Camel. These design patterns document and formalize tried-and-tested approaches to integration that have been harvested from real-world solutions, which together form an integration pattern language we can refer to.

Hohpe and Woolf distinguish four main integration styles in EAI, which can be considered top-level approaches on which integration patterns are based. These are File Transfer, Shared Database, Remote Procedure Invocation and Messaging. While all four integration styles are widely used as part of an integration architect's toolkit, messaging is the most powerful of the four given its loosely coupled and asynchronous nature - as such, messaging is the only integration style that is covered by Hohpe and Woolf's integration patterns and offered in EAI frameworks such as Apache Camel. In this section, we'll go through each of the integration styles and touch briefly upon two common messaging integration patterns: the point-to-point and publish/subscribe messaging.

3.1.1 File Transfer

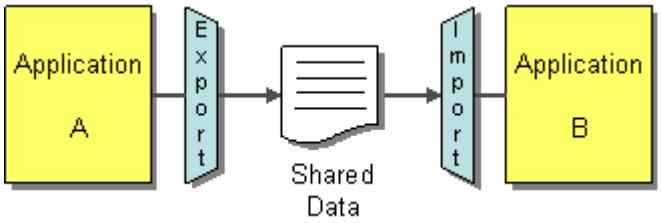

Figure 7. Diagram of the File Transfer integration style (Hohpe & Woolf 2003)

In File Transfer integrations, each application produces files or shared data for others to consume and consumes files that other applications have produced. Integrators take the responsibility of transforming the files into a format that is both readable and processable by the consuming system. These tend to be COBOL file system formats for mainframe systems, text-based files such as CSV for Unix systems, and XML or JSON for more modern applications.

File Transfer integration is useful for importing and exporting tabular data, particularly when integrating with legacy systems which might not have APIs. It is often used for

loading data into relational databases through CSV files, a format that is widely used by end users to manage data in office applications such as Microsoft Excel. In the era of web-based enterprise software, File Transfer integration is typically accomplished by setting up a Secure File Transfer Protocol (SFTP) server or a cloud object storage service such as an Amazon S3 bucket, on which other applications can upload their files in the specified format. These files can then be ingested into the application by invoking a trigger or through a scheduled CRON process.

3.1.2 Shared Database

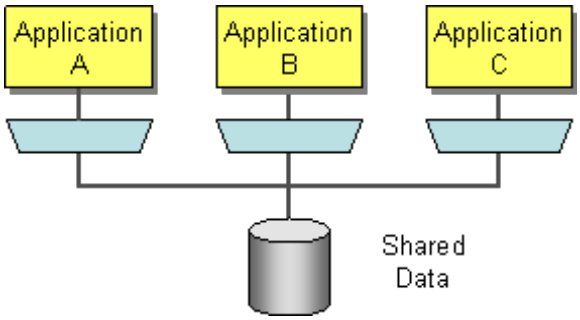

Figure 8. Diagram of the Shared Database integration style (Hohpe & Woolf 2003)

Shared Database integrations ensure that applications are always consistent by having them utilize the same underlying database. This is usually facilitated by a database transaction management system which ensures that simultaneous updates to the same piece of data are handled gracefully through rollbacks and error handling. The governing principle in this approach is centralization: Shared Database integrations aim to achieve consistency between applications by enforcing a single source of truth.

This is the integration style used internally in the software architectures of large ERP and CRM enterprise applications such as SAP, Salesforce, and other large enterprise SaaS products. In the context of application integration, a shared database between several applications provides a powerful way to enforce data consistency, but also creates tight coupling between applications which can be detrimental to both development efficiency as well as runtime performance.

3.1.3 Remote Procedure Invocation

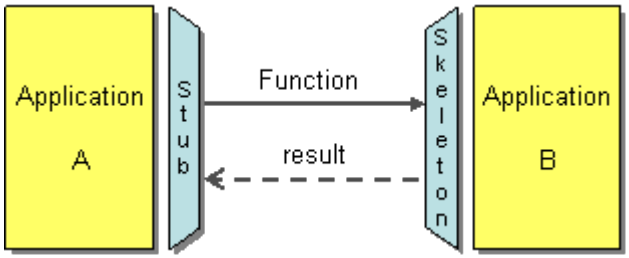

Figure 9. Diagram of the Remote Procedure Invocation integration style (Hohpe & Woolf 2003)

In Remote Procedure Invocation (RPI), each application is developed as a large-scale object or component with encapsulated data but exposes some of its procedures through an interface through which these procedures can be remotely invoked by other applications. The governing principle behind this integration pattern is encapsulation: each application maintains the integrity of the data it owns and can alter its internal data without affecting every other application. If an application wants to retrieve data from - or modify data in - another application, it must make an explicit call to that application.

This integration style has its roots in request-response protocols dating back to the early days of distributed computing of the late 1960's but was first formalized as *Remote Procedure Call* in 1981. It has since been used for distributed computing protocols such as the Network File System (NFS), Java Remote Method Invocation (Java RMI), and early web service protocols such as JSON-RPC and JSON-XML. In more recent years, the RPI style has been implemented in technologies commonly used for modern web services such as SOAP, REST and gRPC.

RPI, by way of web service APIs, has evolved to become the de facto computing interface between software intermediaries in the world wide web. Today, most applications have web service APIs that are available with HTTP via protocols such as REST and SOAP. One drawback, however, is that RPI requires both applications to be up and ready at the same time. If A sends data to B, but B happens to be down, the transaction will fail and cause problems in A's runtime as well as for the system's data consistency. This is particularly problematic if the transaction between A and B is a link in a larger chain of transactions involving other applications. Complex logic for throttling, error handling and fallback procedures need to be developed to handle these scenarios, which adds to development cost as well as complexity.

3.1.4 Messaging

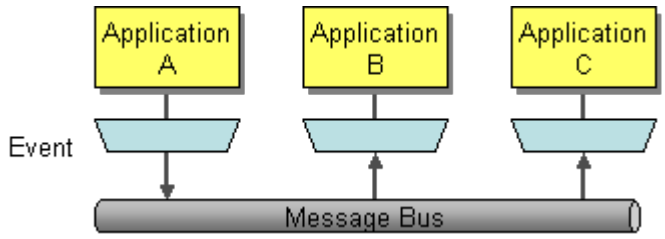

Figure 10. Diagram of the messaging style (Hohpe & Woolf 2003)

Messaging integrations allow applications to communicate with each other in a fault-tolerant, loosely coupled way that counters some of the complications highlighted in the other integration styles. Each application is connected to a common messaging system and uses messages to exchange data and invoke behavior. If Application A needs to invoke a process in Application B such as transferring data for ingestion into a data warehouse, for example, all A needs to do is send a message to the common messaging system which is then received by B through a polling- or event-driven consumer service. This allows data to be transferred “frequently, immediately, reliably, and asynchronously, using customizable formats”. This is the most popular and generally preferred integration style in EAI.

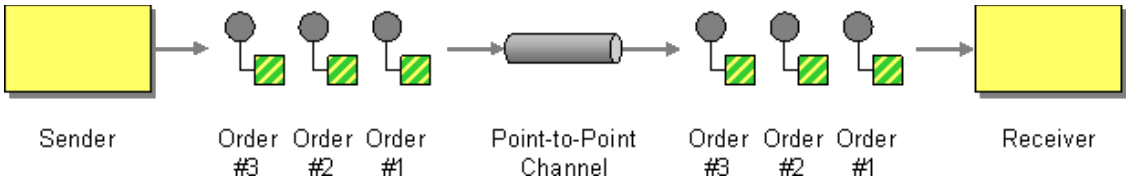

Figure 11. Point-to-Point Channel (Hohpe & Woolf 2003)

Messaging integration patterns generally involve the use of Message Channels. Each messaging system has a set of Message Channels through which data can be transferred from one application to another, and these can be either point-to-point or publish-subscribe channels. In a point-to-point channel, only one receiver consumes any given message - the message is read exactly once and by exactly one receiver, after which it disappears. In a publish-subscribe channel, a single message from an input channel (aka publisher) is split into multiple copies and delivered to multiple output channels, each of which has only one receiver (aka subscriber) and can only consume a message once.

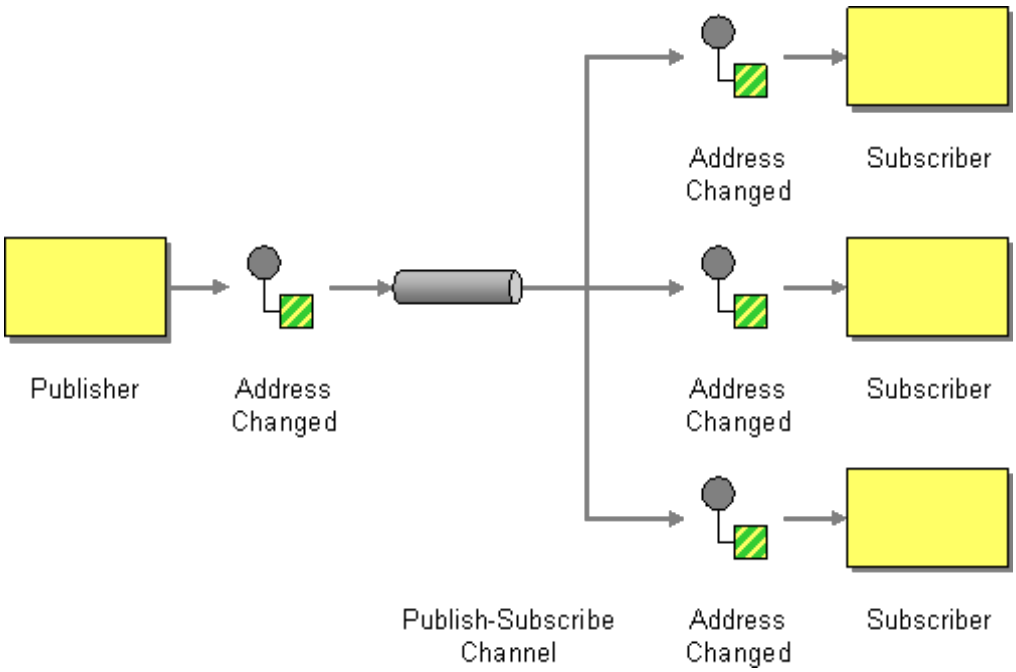

Figure 12. Publish-Subscribe Channel (Hohpe & Woolf 2003)

The use of these two main types of messaging channels allows architects to construct more advanced messaging integration patterns, such as *pipes-and-filters*, in which a series of point-to-point channels are used as "pipes" to chain together independent processing steps or "filters"; and *scatter-gather*, which sends a message through publish-subscribe channels to multiple receivers and re-aggregates the responses back into a single message through a point-to-point channel.

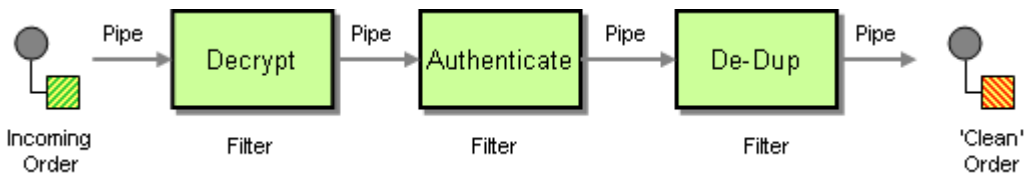

Figure 13. Pipes and filters (Hohpe & Woolf 2003)

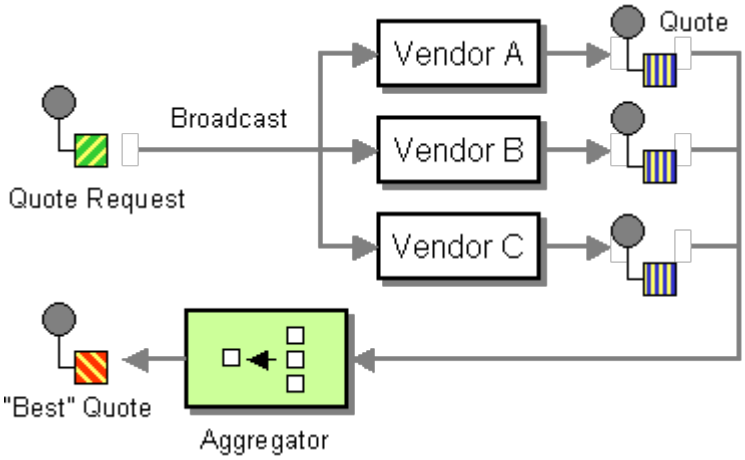

Figure 14. Scatter-Gather (Hohpe & Woolf 2003)

3.2 Integration Architectures

Enterprise integration patterns are often mixed-and-matched to accomplish the various integration needs that emerge in an enterprise system. A file transfer integration, for example, can be used in combination with messaging to trigger processing workflows once a file has been uploaded to an SFTP server or object storage bucket. As the number of integrations in a network grows, however, the need to manage the topology of integrated applications becomes more and more important to keep complexities and costs in check. This is the realm of enterprise architecture, which we will explore in this section.

3.2.1 Point-to-Point

The most straightforward way to integrate multiple applications into a single unit is through point-to-point integration (also known as *one-to-one integration*). This involves connecting two or more applications using an interface between each application, where data is transformed into a compatible format at each end. Point-to-point integration is sometimes seen as an attractive option as it is lightweight and requires minimal overhead, especially in the case of greenfield or small implementation projects where only a few applications need to be integrated. For such projects, using point-to-point integration can keep microeconomic (project) costs low. However, this comes with a number of hidden costs which become apparent as the number of integrated applications in an organization increases. Most organizations are increasingly seeing the need to rapidly adopt new IT systems, and in these cases the point-to-point model can quickly become unmanageable and fragile. This is illustrated by Salesforce.com's point-to-point integration architecture circa 2014, during which they were growing at 30% year over year and had to integrate a number of development environments, back-office systems and software acquisitions.

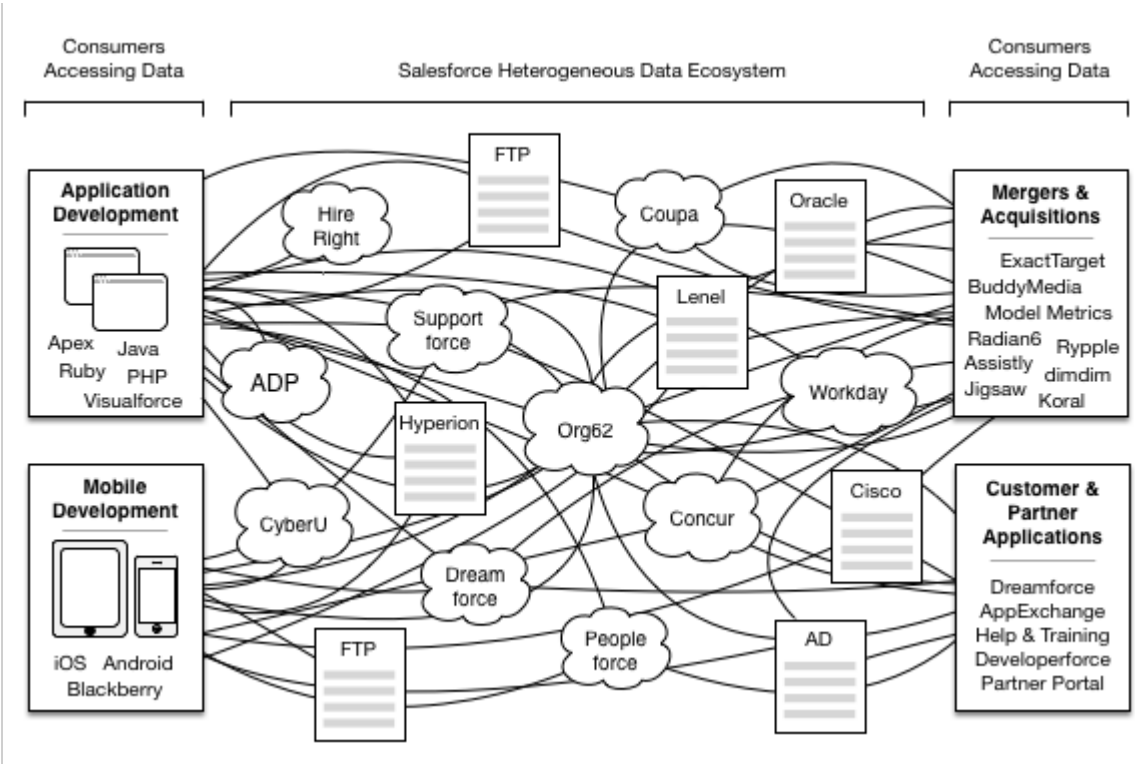

Figure 15. Diagram of Salesforce.com's point-to-point integration architecture circa 2014. (Tiernan 24 November 2014)

There are several main reasons why point-to-point integrations are not feasible for EAI. First, the complexity of any point-to-point integrated system scales exponentially because each integrated application must communicate with every other application independently. For an architecture consisting of *n* applications, the number of interfaces required to integrate all applications is *n(n-1) / 2*, meaning that the number of integration interfaces increases at an exponential rate relative to the number of applications.

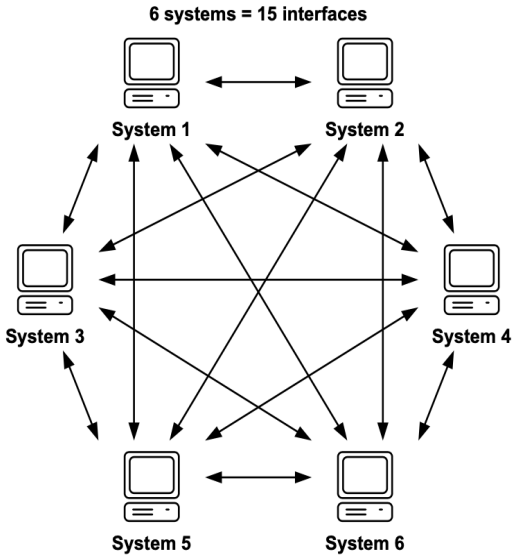

Figure 16. Illustration of number of interface connections in an architecture containing 6 systems. (Koreans 2014)

For instance, an architecture containing three applications requires only three interfaces for full integration. With six applications, however, the number of interfaces quickly increases to fifteen - and at eight applications to twenty-eight interfaces. For any architecture consisting of more than three applications, therefore, the number of interfaces scales exponentially, which can cause development and maintenance costs to spiral out of control. Furthermore, as the marginal value of each new application on average only scales linearly while the cost scales exponentially, each additional component incurs greater marginal cost.

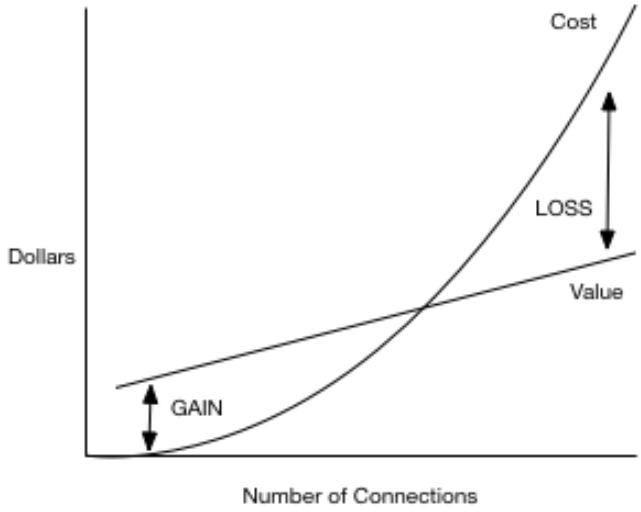

Figure 17. Each additional component in a point-to-point network incurs greater marginal cost (Tiernan 24 November 2014)

Given the tendency of point-to-point systems to degrade into what is referred to as a “big ball of mud” - i.e., systems that lack perceivable architecture or order - various integration architectures have evolved to manage the complexities emerging from the topologies of integrated applications. Chronologically, these are hub-and-spoke, service-oriented architecture and microservices, each of which has built and expanded upon their predecessors.

3.2.2 Hub-and-Spoke

Also known as a message broker, hub-and-spoke architecture emerged as an inevitable reaction to the exponential scaling of point-to-point topologies. By introducing a central hub through which all applications communicate, each application only needs one interface to integrate with the network - meaning that the integration complexity of the network can theoretically scale linearly with the number of applications (Kökörčený 2014).

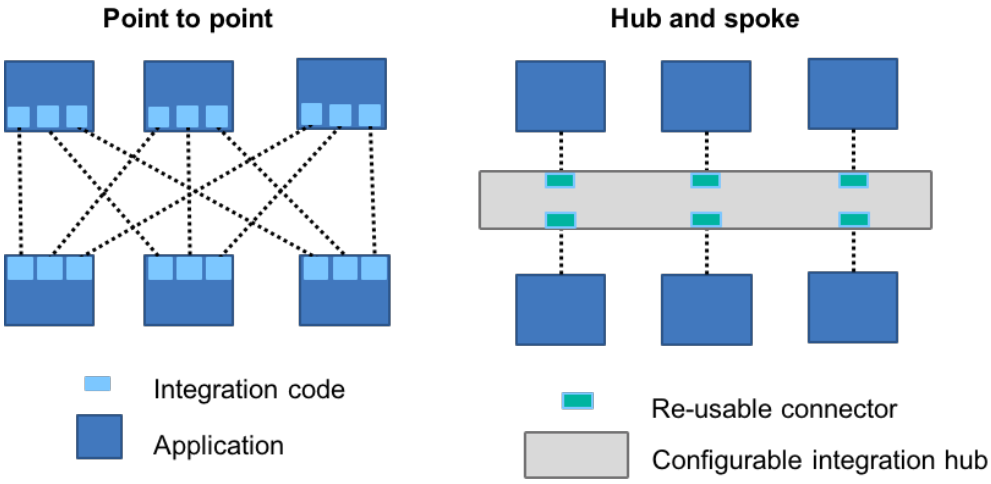

Figure 18. From point-to-point integrations to hub-and-spoke (Clark 2018)

Hub-and-spoke architecture usually consists of message-oriented middleware in which communication between the hub and the spokes is achieved using message channels such as point-to-point (i.e., one-to-one) or publish-subscribe (i.e. a one-to-many or “fan-out”) implemented by means of a middleware layer. In enterprise integration, hub-and-spoke architecture is thus primarily asynchronous and typically designed for integrating on-premises, back-office enterprise applications, with a major advantage over point-to-point being that the hub decouples applications from one another, and integration middleware can be reused.

A major problem with this approach, however, is that the hub still needs to provide protocol translation, where a message received from a message queue needs to be passed on using HTTP, and data transformation, where data is mapped to a model compatible with the target application. If one were to add a message translator between each application in a hub-and-spoke architecture, then one would end up going back to point-to-point architecture since the number of translators scales exponentially with the number of applications.

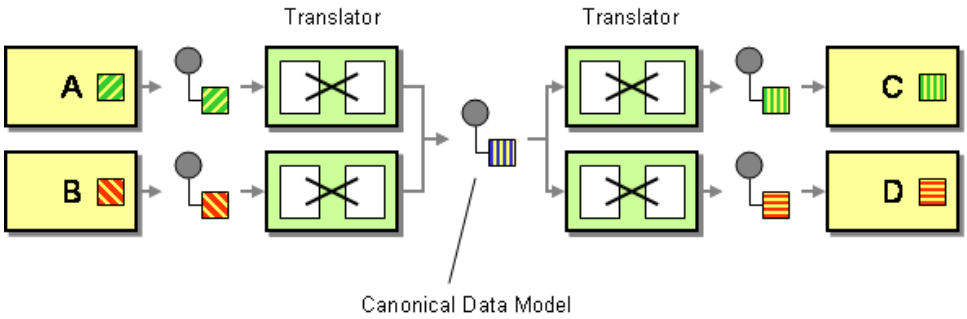

Figure 19. Canonical Data Model (Hohpe & Woolf 2003)

One solution to the problem of data formats, proposed by Hohpe (2003), is to use what he terms a Canonical Data Model - that is, impose a data model that is independent from any specific application and require applications to produce and consume messages in this format. Having a data format general enough to be used by all applications can be very challenging.

3.2.3 Service-Oriented Architecture

As the internet became ubiquitous during the 2000s and the standards HTTP, XML and SOAP became widely accepted, Service-Oriented Architecture (SOA) emerged as a way to expose on-premises, back-office enterprise applications (i.e. systems of record) as a suite of reusable services through standard network protocols. SOA builds upon hub-and-spoke architecture by having the hub also serve as an interface exposing functions from its spokes as re-usable services (IBM Cloud Education 2021). We can therefore view SOA as consisting of two interoperating patterns: asynchronous hub-and-spoke in the background and synchronous service exposure in the foreground.

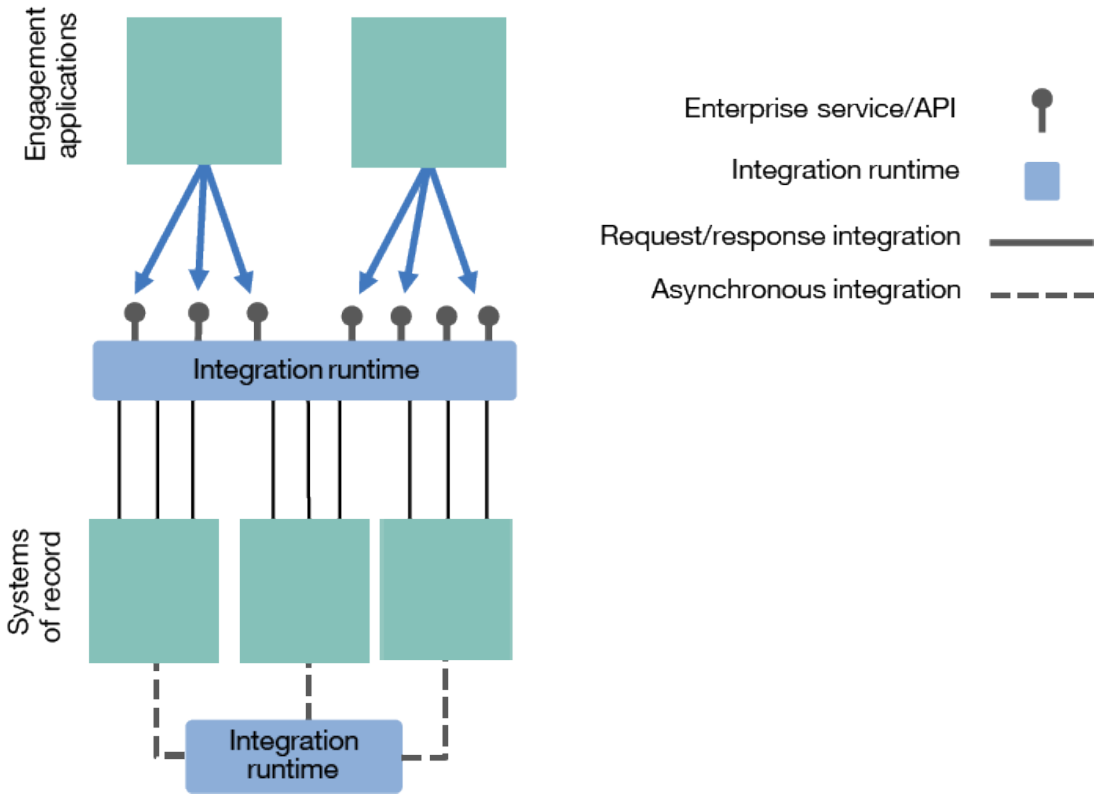

Figure 20. SOA pattern (Clark 2018)

Enterprise Service Bus (ESB) became the dominant way of implementing SOA and is available through both commercial products such as Mule ESB, Oracle Service Bus, Microsoft BizTalk, IBM App Connect and TIBCO, as well as open-source software such as Apache Camel, Apache ServiceMix, JBoss ESB and Spring Integration. Despite the

availability of these tools, ESBs - especially older commercial products - turned out to be quite challenging to develop and maintain perhaps due to the large, enterprise-wide scope of SOA. Given that one of the core benefits of SOA is the reusability of interfaces, it made sense to centralize ESB development so that integrations can be reused across all enterprise applications and projects. This resulted in the phenomenon where organizations often end up with a single ESB infrastructure supporting the entire enterprise, along with a dedicated team of specialists responsible for developing and maintaining it.

The centralized ESB pattern created several problems that are characteristic of monolithic design. First, integrations become tightly coupled and therefore carry high interdependency risks - changes deployed to one interface often destabilize unrelated interfaces, resulting in the need for complex regression testing as well as resulting in high disincentives to releasing software. Second, due to the large number of integrations they contained, centralized ESB runtimes are often very large and caused significant downtimes during deployments, thereby encouraging a preference for live-patching hot fixes. These further compounds disincentives to releasing software, as ad hoc changes and server configurations make it difficult to replicate environments for diagnosis and testing. Third, as ESBs typically must integrate with legacy applications that do not have easy-to-use interfaces, the integrations are often highly complex and require expertise that only a small number of integration specialists can provide. This not only dictates the need for a specialist integrations team that operates the ESB in a Waterfall style, but also means that filling vacancies in the team is often challenging due to a scarcity of suitable candidates.

3.2.4 Microservices

Microservices emerged as a way to tackle the challenges resulting from centralized, monolithic design. While SOA and ESB have an enterprise scope focusing on integrations between different applications, the microservice architecture emerged from within the context of internal components within applications. In the years prior to the rise of microservices, application architectures gravitated towards large, centralized monoliths, and just as centralized ESBs had become bloated and difficult to maintain, so too had applications. Monolithic applications, consisting of interdependent components packed into a single-tiered structure, quickly became costly and fragile, encountering many of the same problems that centralized ESBs faced. To counter these challenges and meet the growing need for IT organizations to be agile and scalable, developers started breaking applications down into smaller units which can be developed and run independently. This

eventually evolved into an architectural style where an application is structured as a collection of services that are as follows (Richardson 2020):

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

Applying microservices principles to EAI means splitting up the centralized ESB into smaller, independently maintainable, and easily scalable pieces. Just as in SOA, the microservice architecture's integration runtime consists of a both an exposure gateway through which a suite of reusable services can be accessed as standard network protocols, as well as asynchronous back-office integrations between systems of record. The way these are implemented, however, differ greatly from ESB.

First, thanks to the standardization of REST as a network protocol and the resultant emergence of API gateways and management, applications no longer need an ESB's integration runtime as an intermediary to translate communication protocols - instead they can publish directly to an API gateway through which it can be discovered. With the rise of virtualization technologies such as Docker, orchestration frameworks such as Kubernetes, developer tools for continuous integration and deployment, and the increasing adoption of cloud computing, integration runtimes have become a lot more streamlined, user-friendly, and cost-effective. Application teams can build lightweight integration runtimes using containers and deploy them through automated continuous delivery processes into cloud environments. These technologies have proven to be key enablers for the decentralization of integration ownership away from specialist SOA teams to the application teams themselves.

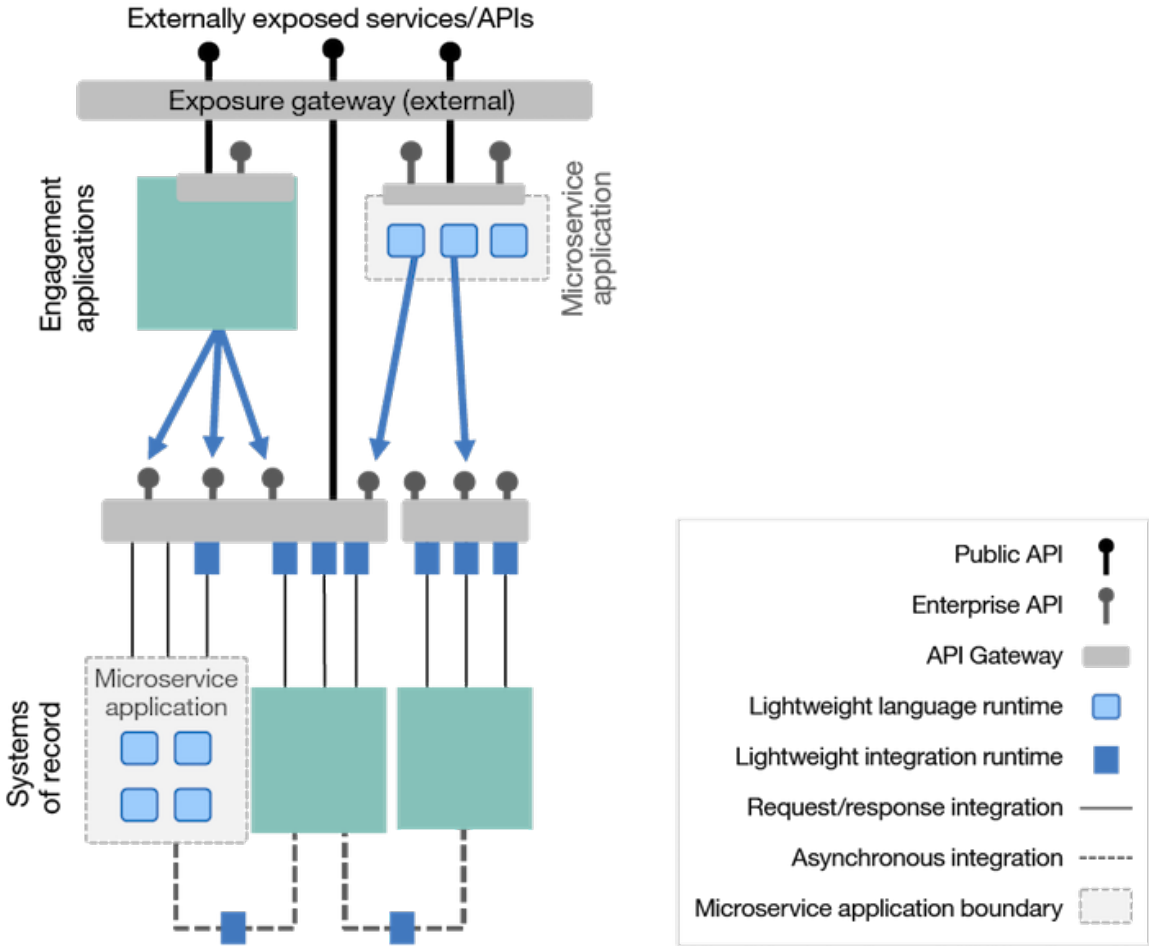

Figure 21. Microservice architecture (Clark 2018)

Second, in microservice architecture, back-office asynchronous integrations between applications no longer rely on a hub-and-spoke pattern, but instead communicate with each other in a decentralized manner. While ESB solves the problem of exponential scaling from point-to-point integrations by providing a single, centralized hub as a message broker, in microservice architecture applications have the freedom of choosing the implementation and scope of their integrations. The key here is choice: if all applications in the enterprise network are deeply integrated with each other, for example, then it might indeed make sense to implement a message or event bus serving as a broker like the hub-and-spoke model. More frequently, however, integrations are between clusters of applications, sometimes involving nothing more than a unidirectional notification message from one application to another. In these cases, the integration work required - such as sending a message to another application through a point-to-point messaging channel - can be trivial to implement as a microservice and can be easily maintained by the application team. This is greatly enabled by the standardization of lightweight protocols such as HTTP and REST as well as the huge improvements in integration tooling. In more-complex integrations involving clusters of applications, applications have a growing suite of integration tools available that provide, for example, message routing and other orchestration capabilities. These are readily available both as

part of PaaS cloud computing environments on which many modern applications live, such as Amazon Web Services, Google Cloud Platform and Microsoft Azure, but can also be implemented as their microservice components. When deployed as a microservice component, integration frameworks with ESB-like capabilities such as Apache Camel implement common integration patterns that enforce best practices and are highly configurable, which can be a lightweight alternative to traditional centralized ESBs which application development teams can themselves maintain.

Using microservices for EAI is therefore a huge improvement over centralized ESB as it provides greater ease and flexibility, which makes the development and maintenance of integration runtimes less costly and more agile. Additionally, the lightweight nature of microservices opens the possibility for decentralized management of the integration runtime, in which the application development teams themselves can maintain their application's integration runtime within decentralized clusters of applications in an enterprise system.

3.3 Serverless Computing

3.3.1 Technological Background

We have seen that microservices emerged in reaction to the challenges posed by monolithic applications and ESBs, and this has been enabled in large part by a few key technologies. The most important of these technologies is containerization, particularly the open-source Docker Engine, which allows application code to be easily packaged along with its dependencies and reliably deployed in different computing environments. Combined with the adoption of cloud computing, continuous integration tools and standardization of communication protocols, containers have enabled development teams to break down services into modular, independent components without worrying too much about development overhead and economies of scale. Containers accomplish this by virtualizing the operating system on which they are hosted, allowing multiple containers to share the same machine and OS kernel while running in isolation from each other. Containers are therefore like virtual machines in that they are both a means of providing resource isolation and allocation through virtualization. But whereas virtual machines are an abstraction of physical hardware, containers are an abstraction at the application layer.

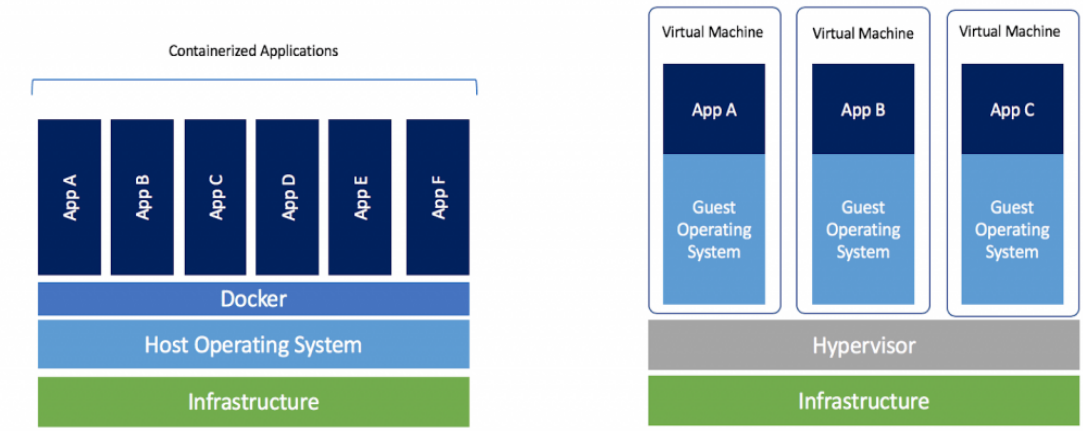

Figure 22. Comparing Containerized Applications to Virtual Machines (Fong 2018).

Serverless computing builds upon containers by providing an even further layer of abstraction at the level of application functions. While containers allow developers to quickly allocate computing resources and spin up application servers, in serverless computing the cloud service provider allocates computing resources on demand whenever a function is executed, thereby abstracting away the concept of a server from the application entirely (hence its name). Serverless computing is somewhat of a misnomer as physical servers still exist and are managed by the cloud service provider, but all aspects of this management - such as capacity planning, scaling and virtual machines - are either completely hidden or easily configurable by the developer. Furthermore, serverless computing utilizes a pay-as-you-go pricing model where costs are incurred solely based on the time and memory allocated to run the application code, with no costs for idle time. When compared to explicitly provisioned servers which generally have periods of idle time during which resources are underutilized, serverless computing has the potential to bring significant cost savings, particularly in cases where usage is low.

3.3.2 Serverless versus Containers

Without going into too much detail comparing serverless with container-based microservice architectures, it suffices for now to say that serverless architecture automates infrastructure operations to an even greater extent than container-based microservice architectures by handing over control of infrastructure operations to the cloud computing provider. As we shall see, this comes with both benefits and drawbacks. With the help of technologies such as Docker, Kubernetes and Helm, system administration for containerized microservices (such as CPU, memory, and networking) can be configured precisely to business needs and deployed to a wide range of server hardware including all the major cloud computing platforms. There are also technologies which provide autoscaling and maintenance automation capabilities for containerized microservices

which enable them to achieve some of the resource-optimization benefits offered by serverless. Containerized microservices are therefore still the go-to computing infrastructure of choice for many organizations, particularly for databases, stateful applications, and workloads which have consistent, predictable throughput especially when large data volumes or memory usage is involved.

In the traditional waterfall model of software engineering, completed application code is handed over from a software development team to an operations team, who are then responsible for deploying the code and overseeing its subsequent operations such as network, security, and server administration. As discussed, the automation capabilities offered by microservice containerization technologies are thus a huge improvement to this previous model, to the extent that the traditional system administration tasks are quickly being automated away. Docker and Kubernetes enable software developers to do many of the tasks traditionally reserved for system administrators, thereby dramatically improving the cycle time and scalability of software releases. However, the tooling needed to continuously integrate and deploy these containers, as well as the maintenance operations they require once they go live, are still complex enough that they often still need to be managed by dedicated personnel, typically as part of so-called DevOps or Site Reliability Engineering (SRE) teams. Thus, container technology doesn't fully solve the problem of the disconnect between development and operations - it simply improves it by enabling operations to be more automated, scalable, and closely integrated with the software development process.

One can therefore view serverless computing as an attempt to go one step further than containerization in operations automation. With servers abstracted away and system features such as deployment, security, logging, monitoring, autoscaling and networking provided out of the box by the cloud computing provider, software development teams can in many cases take full ownership of deployment and maintenance operations themselves. For example, instead of needing an operations team to manually scale servers up and down based on usage (or scripting automation capabilities to so), serverless computing orchestrates this process automatically based on usage and charges the cost correspondingly. Every function execution has dedicated memory and CPU cores (up to a certain extent, e.g., ranging from 128 MB to 10,240 MB in AWS Lambda) meaning that computing power is virtually unlimited if memory usage of individual workloads is within the provisioned limits. As mentioned, this is not suitable for some types of workloads such as stateful applications and databases. However, it is ideal for stateless workloads such as data processing, REST APIs and most web application use cases.

3.3.3 Applications in EAI

Given that EAI mostly involves the stateless transformation, translation, and transfer of data between applications, it therefore makes sense that serverless should be a good fit for implementing EAI workloads. However, one crucial element that serverless computing must be able to support is asynchronous messaging, which has been described in the integration patterns in previous sections. We have already discussed Lambda functions, which take an input event such as a HTTP request, execute the function logic and return the result as an output (for non-void functions). These can be used in EAI for performing transformations and business logic. But how is serverless computing used in the context of crucial infrastructure for messaging channels, topics, and queues?

In a microservice architecture approach to EAI, messaging infrastructure can itself be deployed as a microservice component that interacts with integrating applications. Open-source software such as Apache Kafka, Apache Camel, and RabbitMQ all provide a range of messaging features that can be deployed as stateful containers and used to implement the orchestration aspects of enterprise integration patterns. For instance, these messaging systems, deployed as microservices, can be used in conjunction with serverless Lambda functions by providing the central messaging hub through which each integrating application is connected as a node, with a serverless lambda function serving as a transformation and translation layer between the messaging hub and each application. This approach relies on containerized microservices to implement the core EAI messaging infrastructure, with serverless lambda functions as an ancillary component.

While messaging hubs deployed as microservices effectively replace ESBs, they can become too complex in practice for development teams to maintain by themselves. Just as containerized applications often require a DevOps or SRE team to maintain the CI/CD pipelines and administer Kubernetes clusters for example, these messaging hubs may make it again necessary for a dedicated team of specialists to oversee their deployment and maintenance. Therefore, a purely serverless approach to EAI should preferably not have to deal with deploying and maintaining a separate messaging hub. Serverless computing’s answer to this is in the form of lightweight, managed services that can be operated using an on-demand model in much the same way as Lambda functions. These managed services are offered by all major cloud computing platforms, providing messaging services that can be provisioned on-demand with minimal configuration and operational overhead, and can be easily integrated with serverless functions. Since we are focusing on AWS in this thesis, the messaging services we explore are Simple

Notification Service (SNS) and Simple Queue Service (SQS) which we will discuss in greater detail.

3.4 Discussion

To summarize this section, we saw that in the absence of any explicit integration architecture the *de facto* state of an enterprise system can be described as consisting of a network of point-to-point integrations between applications. Using our distinction between vertical hierarchies and horizontal networks, we see that point-to-point integrations represent a maximally horizontal network structure with minimal hierarchical coordination. The nodes in this topology are fully decentralized, such that each node needs to explicitly establish a connection to each other node to share data, business rules and processes. This creates what we referred to as the *exponential scaling* problem, where the number of interfaces needed scales at an exponential rate relative to the number of applications in the network. As a result, point-to-point integrations not only result in a “big ball of mud” in the enterprise system’s topology, but also incurs a number of costs.

Recalling our abstract model of EAI architecture - as a structure that *economizes on the production, coordination and vulnerability costs of organization while minimizing negative externalities incurred from search, decision-making and enforcement* - we can interpret the costs involved in point-to-point integrations as resulting from the transaction costs of market mechanisms. As each application must transact with every other application without a coordinating central hub, point-to-point integrations greatly resemble decentralized markets. As such, point-to-point integrations incur high transaction costs - each node must first locate and connect to every other node on the network (search and information costs), establish protocols and transformation logic for sharing data and business processes (bargaining and decision-making costs), and monitor and protect the deployed integration runtimes (enforcement and policing costs). From the perspective of transaction cost economics, therefore, we can describe the costs of exponential scaling in point-to-point integrations as negative externalities resulting from the transaction costs of market mechanisms.

Table 10. Comparison of EAI architectures in terms of transaction, production, coordination, and vulnerability costs.

	Transaction Costs (search, decision-making, enforcement)	Efficiency	Flexibility	
		Production Costs	Coordination Costs	Vulnerability Costs
Point-to-Point	-	-	-	-
Hub-and-Spoke	+	-	-	-
SOA	++	+	-	-
Microservices	+	+	+	++
Serverless	++	++	++	+

- does not economize

+ economizes

++ economizes to a greater extent

Just as hierarchical firms emerge to economize on the transaction costs of exchange in markets, so too do hierarchical integration architectures emerge to economize on the transaction costs of point-to-point integrations. Hub-and-spoke and service-oriented architecture impose a high degree of hierarchy upon a network of applications in order to lower the costs of search, decision-making and enforcement, and hence present a maximally hierarchical organizational form. Hub-and-spoke can be compared to a basic hierarchical firm - all integrating applications simply connect as spokes to the hub and communicate with one another through a centralized message broker. Although hub-and-spoke integrations mitigate the transaction costs of market exchange (point-to-point integrations) by bringing them under the control structure of a firm (centralized integration hub), it neither achieves efficiency nor flexibility when compared to other integration architectures.

Service-oriented architecture, on the other hand, presents a more sophisticated form of hierarchical organization which can be described as U-form or *efficient* hierarchical structure. By encapsulating software components into specialized functions, SOA economizes on *production costs* to a much greater extent than the hub-and-spoke architecture. Functionality offered by implementations of SOA such as ESBs further economizes on transaction costs by offering features such as an exposure gateway over which the reusable services can be found and accessed. However, its tight coupling of interdependencies, centralized release orchestration and high degree of skill specificity means that it carries great *coordination* and *vulnerability costs*, and therefore it sacrifices flexibility in favor of its efficiency.

Microservice and serverless architectures revert the hierarchical pattern of SOA by breaking it down into a more decentralized structure (i.e., horizontal network), thereby economizing *coordination* and *vulnerability costs* and achieving greater flexibility. Loose coupling, self-organization, and independence all lower coordination and vulnerability costs as they enable decision-making autonomy, shorter cycle times, greater levels of experimentation, and quicker adaptation to technological shock. Microservices and serverless can therefore be described by as horizontal network structure, or the M-form organization in Malone's model. However, while the M-form's flexibility should come at the expense of relatively greater production costs, this is not clearly the case with microservice and serverless architectures. Indeed, splitting up the centralized ESB into smaller, independently maintainable microservices requires setting up more dedicated computing environments, and it might seem that microservices should therefore incur higher production costs. However, as virtualization, cloud computing and continuous integration technologies continue to improve, the production costs of a microservice architecture have greatly decreased to the point that they can out-compete SOAs.

This is even more pronounced in serverless architecture. With its pay-as-you-go pricing model, serverless computing has extremely streamlined production costs, making it a much more efficient option than microservices for many types of workloads. Another key advantage that serverless has compared to containerized microservices is the fact that it offers full application functionality out of the box. These features help lower the costs of coordination for serverless integration architectures, decreasing the amount of manual configuration, back-and-forth communication and other types of explicit coordination that is typically required between microservice teams.

As we discussed earlier, however, the decentralized network structure embodied by microservices and serverless have become so highly integrated that the distinction between market and firm becomes blurred, meaning that market mechanisms are again effectively an integral part of the organizational structure of firms and along with them the transaction costs (i.e. negative externalities) of search, decision-making and enforcement. In order to mitigate these negative externalities, teams need to develop administrative tools for tasks such as service discovery, deployment, security, audit logging, and monitoring, and the costs they incur are often non-trivial. Without such tools, we would find ourselves in the realm of point-to-point integrations again, with a big ball of mud and crippling transaction costs. This is where serverless architecture also has a significant advantage over containerized microservices - its full application functionality means that these administrative tools are offered out-of-the-box and can be easily configured in UI dashboards, templates or via the cloud platform's API and SDKs.

One area in which serverless computing is at a disadvantage compared to microservices, however, is vulnerability costs. In general, developing services using cloud computing always has inherent risk of vendor lock-in, where an application or architecture has become so specific to a certain cloud vendor that it becomes costly to migrate away from the vendor and explore other options (hence being “locked-in” to a certain vendor). Because vendors usually offer a range of choices in technology (e.g., operating systems, programming languages), they still enable a great deal of flexibility and operate very well with open source - hence, the vulnerability costs in serverless are not as great as those of ESBs, for example. However, serverless implementations inevitably require some amount of vendor-specific code, which imposes vulnerability costs in situations where the organization must migrate away from the vendor’s cloud infrastructure. Containerized microservice architectures, on the other hand, can encapsulate software in containers such that explicit infrastructure assumptions are minimized.

To summarize, based on our theoretical framework, serverless computing presents an attractive approach to EAI. It is able to economize on production costs through pay-as-you-go pricing, coordination costs through full application “as a service” functionality, and vulnerability costs through supporting a wide range of open-source technologies that enable experimentation and adaptation. While the risk of vendor lock-in makes it slightly more vulnerable than microservices, it has an advantage over containerized microservices in its ability to mitigate the negative externalities of decentralized integrations through out-of-the-box functionality. In the following sections, I will present a design and implementation of a serverless EAI architecture to how well it holds up to our theoretical evaluation of it.

4 Case Design

In this case scenario, we are tasked with building backend services for a company's customer loyalty program on top of its newly acquired CRM system. Salesforce has been selected as the new CRM for loyalty member data such as accounts and contacts and serves both as the back-office sales and customer service tool used by internal employees, as well as the "master data" system of record used by customer-facing digital channels. We can assume that front-end applications such as a website and mobile application already exist, and therefore what is needed are the backend services for connecting these digital channel applications to Salesforce. Since several other applications are also used as part of the loyalty program's information system, they also need to be integrated with Salesforce.

4.1 Overview

For the purposes of this scenario, there are four back-office applications used by the loyalty program: a CRM system (Salesforce), a loyalty system (rule engine), an email automation system, and a data warehouse:

- **Salesforce:** Salesforce is the cloud CRM platform that serves both as the "master data" for the organization's contacts and accounts as well as the main GUI application for all sales and customer service personnel to manage interactions with customers.
- **Loyalty System:** This application is responsible for managing each customer's membership after they have enrolled in the company's loyalty program, and therefore stores data such as tiers, points and transactions. It serves as a rule engine that evaluates when a customer should be promoted to the next tier, for instance, or when they should be given an award.
- **Email Automation:** The email automation system is an application used for marketing automation as well as for sending emails in specific user flows such as sign up and deregistration, and therefore relies on customer data such as the customer's name to construct personalized email content.
- **Data Warehouse:** The data warehouse is a data store used as the source of truth for back-office business intelligence and analytics, and therefore ingests data from Salesforce such as contacts and accounts.

Since this scenario is artificial, all integrating applications in this implementation are stubbed except for Salesforce. This means that although they exist and function as integrating nodes in the architecture, they are not real enterprise applications and for the most part do not have fully-fledged functionality. The loyalty system, for instance, does not have a functioning rule engine in this implementation as that is outside of the focus of application integration. Rather, these applications are implemented using either (1) a NoSQL document database or (2) a simple serverless function. These stubs should be sufficient to demonstrate how data and business processes can be shared across several

applications. The connection to Salesforce, which is not stubbed in this case project, contains examples of implementing a commercial application's API which should be representative of what integration code for other commercial applications would look like via REST interface.

For this scenario, we limit the development scope to two customer flows: registering to the loyalty program, and deregistering from the loyalty program. These can be written as two user stories with corresponding acceptance criteria:

(1) **Enroll Member:** As a customer, I can enroll in the loyalty program so that I can earn points and get better more personalized services. Acceptance criteria:
    (a) A new Contact record is created in Salesforce containing customer information
    (b) A new loyalty member record is created in the loyalty rule engine
    (c) A copy of the Salesforce Contact record is ingested into the data warehouse
    (d) A "welcome" email is sent upon successful registration
(2) **Unenroll Member:** As a customer, I can unenroll from the loyalty program so that my data is removed from all systems of record. Acceptance criteria:
    (a) The customer's Contact record is deleted in Salesforce
    (b) The customer's loyalty member record is deleted from the loyalty rule engine
    (c) The data warehouse's copy of the Contact record is deleted
    (d) A "goodbye" email is sent upon successful deregistration

Other customer flows in a typical loyalty program that would require synchronicity between the applications such as updating profile information, processing transactions and upgrading tiers are out of the scope of this scenario. However, integrations for these other customer flows should be able to follow a similar pattern as the one explored here, which exposes integration processes as services accessible via an exposure gateway.

4.2 Infrastructure-as-a-Service

The cloud computing platform Amazon Web Services (AWS) is used to provide the computing Infrastructure-as-a-Service for the implementation of the two user stories above i.e. (1) Enroll Member and (2) Unenroll Member. In particular, we make use of three serverless technologies that are offered as services on the platform: AWS Lambda, Amazon Simple Notification (SNS) and Amazon Simple Queue Service (SQS):

- **AWS Lambda:** This is the serverless compute service that we use as the foundational building block for executing workloads. Lambda functions are ideal for stateless, high-volume, short duration microservices such as REST APIs, and are therefore typically configured to be executed from HTTP requests coming from an

API gateway. However, they can also be configured to execute on other event sources, such as consuming events from a queue or running on a cron schedule.

- **SNS:** Amazon Simple Notification Service is a managed pub/sub messaging service that can be used for both application-to-application as well as application-to-person messaging. In the case of distributed serverless lambda functions, SNS can be used to implement many-to-many messaging between them. A messaging channel consists of (1) an SNS Topic; (2) a service (such as a lambda function) that has permission to publish messages to that SNS topic; and (3) a subscriber (such as a messaging queue, SMS, or email) that reads the messages from the channel.
- **SQS:** Amazon Simple Queue Service is a fully managed message queuing service which is typically paired with SNS to form the receiving end of a messaging channel. A queueing mechanism is necessary in messaging channels in order to decouple publishers from subscribers. SQS queues are therefore typically used in conjunction with lambda functions to subscribe to SNS messages, allowing lambda functions to read messages from SNS asynchronously.

These services provide us with the modular building blocks we need to implement our integration architecture. Although this design and implementation make explicit use of the Amazon Web Services platform, similar serverless computing services are available on competing platforms such as Google Cloud Platform and Microsoft Azure. However, these alternatives are not in the scope of this case's design and implementation, and they may not contain all the necessary services needed to build the architecture proposed in this case.

4.3 High-Level Architecture

To get a high-level overview of the architecture, it is helpful to group components into four layers of logical abstraction: the exposure gateway, engagement layer, integration layer and the back-office layer. As we shall see in the implementation, however, these are not necessarily separated as such at the code- and infrastructure-level. Indeed, this is a feature of serverless design patterns. Whereas a hierarchical integration architecture might have monolithic applications housing each of these different layers of processes, and separate teams managing each, serverless processes are distributed and mostly share the same modular building blocks of lambda functions, document data stores and messaging queues, which means that the physical boundaries of different processes are not as clearly defined as their functional boundaries.

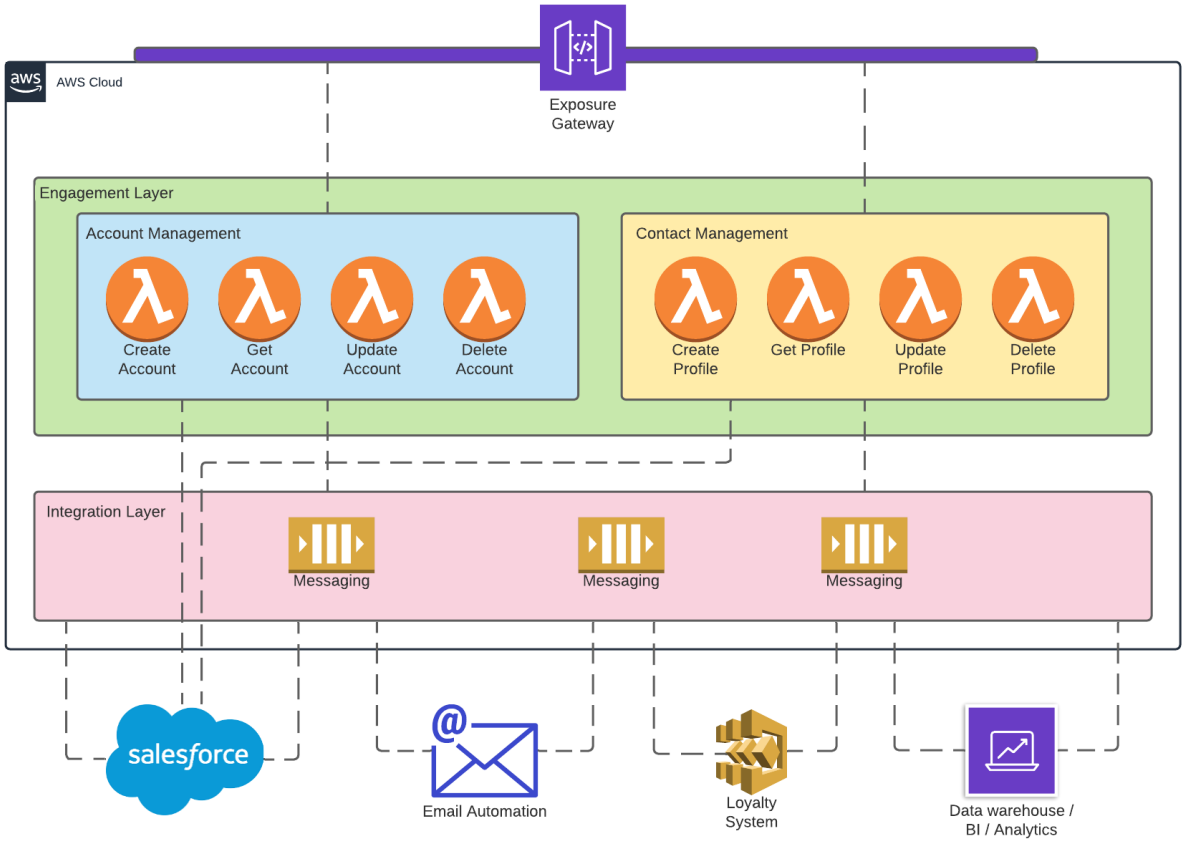

Figure 23. High-level architecture

4.3.1 Exposure Gateway

The Exposure Gateway is the point of access to the loyalty program's backend services from the internet. The internet in this case primarily consists of web and mobile applications in the company's customer-facing digital channels, and these invoke the engagement layer APIs through the exposure gateway via HTTP. To ensure data privacy and mitigate security threats such as distributed denial-of-service (DDoS) attacks, access control should be in place at this layer. Organizations which have high traffic and globally distributed users should also consider using a content delivery network (CDN) such as Cloudflare or Akamai which provide content caching, edge computing, and bot mitigation capabilities that can improve performance as well as security. As these topics are not the focus of this case scenario, however, a simple API gateway protected by API key will suffice.

Each API in the engagement layer is exposed as an endpoint on the API Gateway and can be invoked via HTTP requests. For this scenario, I use a custom domain name https://eai.ceora.app and configured Google's DNS service to route the domain name to my Amazon API Gateway instance, which in turn exposes each of the APIs in the engagement layer as path endpoints. The path https://eai.ceora.app/salesforce/account and its subpaths route to APIs in the *Account Management* module, while the path

https://eai.ceora.app/salesforce/contact and its subpaths route to APIs in the *Contact Management* module.

Figure 24. Amazon API Gateway resource paths

4.3.2 Engagement Layer

This layer is an abstraction of the subset of processes that serve as APIs for customer-facing digital channel applications. The APIs in this layer use the REST protocol for stateless transfer of data over HTTP methods such as GET, POST and PUT. The *Account Management* module consists of the lambda functions related to Salesforce Account records, while the *Contact Management* module consists of lambda functions related to Salesforce Contact records. Each lambda function is implemented as a REST API and is mapped to an endpoint and request in the exposure gateway.

To register a new loyalty member (User Story 1), for example, we can use cURL to make a POST request to `https://eai.ceora.app/salesforce/contact` which is mapped to the Create Profile API.

```
curl --location --request POST 'https://eai.ceora.app/salesforce/contact' \
--header 'x-api-key: <API_KEY> \
--header 'Content-Type: application/json' \
--data-raw '{
   "LastName": "Labadie",
   "FirstName": "Hellen",
   ...
}'
```

Similarly, to fetch an existing customer we can make a cURL HTTP GET request to `https://eai.ceora.app/salesforce/contact/<CONTACT_ID>`, which is mapped to the Get Profile API

```
curl --location --request GET 'https://eai.ceora.app/salesforce/contact/<CONTACT_ID>' --
header 'x-api-key: <API_KEY>
```

In this scenario, the engagement layer serves primarily as a proxy between the internet and Salesforce, fortifying access to Salesforce from the internet while at the same time performing all necessary protocol and schema translations for integrations. For instance, if the loyalty web application's schema differs from Salesforce's, a field mapping can be performed at this layer. Protocol translations (e.g., from HTTP to messaging) are also performed at this layer so that asynchronous integration processes can take place in the downstream integration layer.

4.3.3 Integration Layer

The integration layer is an abstraction representing the processes which perform integration tasks between the back-office systems. The processes in the integration layer implement the integration patterns explored earlier and serve as the bridge not only between the engagement layer and back-office applications, but also for processes among different back-office applications.

For instance, a web service in the engagement layer - say a backend service for a web application - can send a message to a topic *Z* that is subsequently consumed by a back-office application *X* which initiates a business process. Another back-office application *Y*, however, can also send a message to topic *Z* to trigger the business process in application *X*. This is illustrated to highlight that back-office applications can communicate both through the exposure gateway (i.e., with HTTP) or through the integration runtime itself via messaging protocols.

4.3.4 Back-Office Layer

This layer is an abstraction representing the enterprise applications that serve as back-office systems with which the Integration Layer communicates. It can consist of applications that are hosted within the organization's network on the same cloud, on-premises on the organization's own infrastructure, or outside of the organization's network - as is the case with Salesforce which is hosted on its own data centers and only available as a cloud service. For the purposes of this scenario, all back-office applications except

for Salesforce are within the organization's network on the same cloud (since they are implemented using stubs, as discussed earlier).

Back-office applications generally have their own data stores and copy of the data, but this is not always the case as some applications are explicitly designed to be used in conjunction with, or on top of, existing applications. This is how I have intended the Email Automation system to be in this implementation scenario, which has a *Shared Database* integration with the data warehouse (See *Integration Patterns - Shared Database*). The Email Automation system needs to access customer data to construct its email content, and although it could achieve this by maintaining its own copy of the data, in this scenario the Email Automation system needs to fetch customer data from an external source at process invocation, which in this case is the data warehouse.

4.4 Integration Architecture

4.4.1 Enroll Member

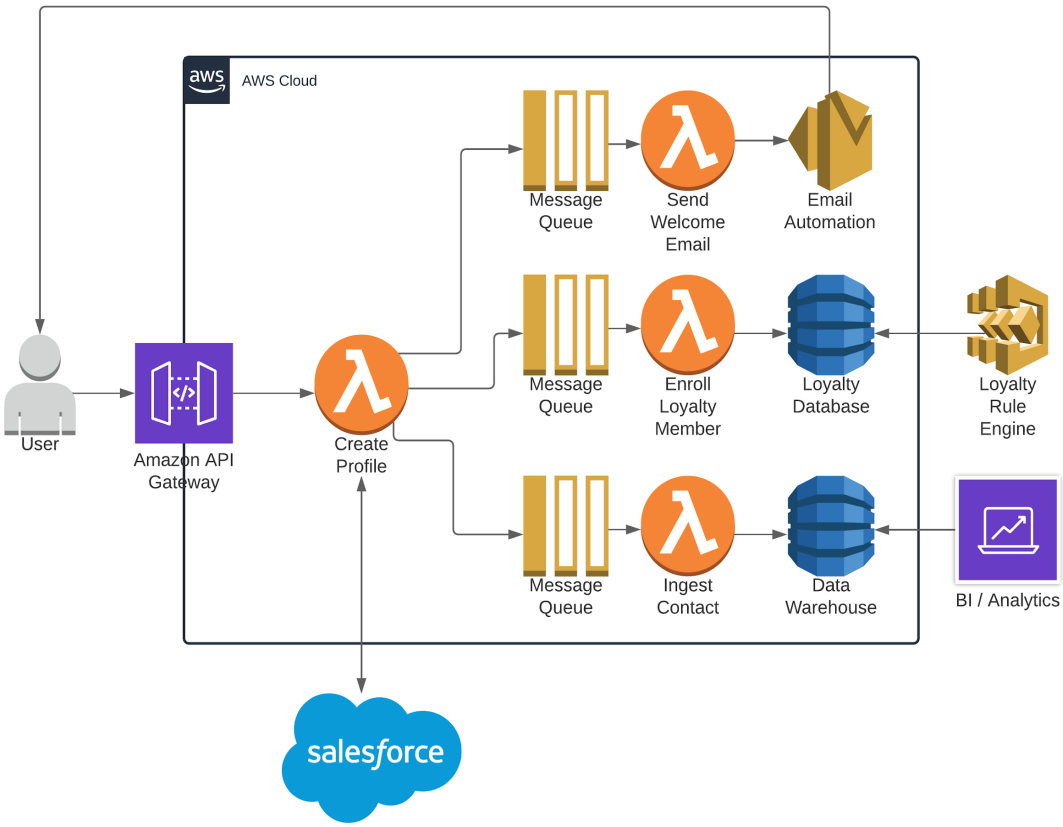

Figure 25. Architecture of integration flow for user registration

The integration flow in the member enrollment process uses a *fanout* messaging pattern, which spreads a message to multiple destinations in parallel. Recalling the messaging

integration patterns in an earlier chapter, the fanout strategy uses a *publish-subscribe* messaging channel. The *Create Profile* lambda function processes incoming requests from the internet routed via the API Gateway and creates a Contact record in Salesforce via its REST API. If Contact creation in Salesforce is successful, the *Create Profile* publishes a message to an SNS Topic which notifies its subscribers that the Contact record has been created, at the same time forwarding the Contact record's data by encoding it in JSON and encapsulating it in the message body.

The *Create Profile* lambda has no knowledge of which processes are subscribed to the SNS Topic - all it needs to do is publish the message with JSON-encoded Contact data to the topic. In this user registration flow, the *Create Profile*'s SNS topic has three subscribers: the *Send Welcome Email* lambda function which sends an email welcoming the newly registered member by invoking the Email Automation system (stubbed by Amazon SES); the *Enroll Loyalty Member* lambda function which creates a Loyalty Member record in the Loyalty System (stubbed by a DynamoDB table), and the *Ingest Contact* lambda function which makes a replica of the Contact data in the Data Warehouse (stubbed by a DynamoDB table).

These lambda functions subscribe to the *Create Profile* SNS Topic using Amazon SQS message queues - each lambda function has its own message queue that serves as a buffer for incoming messages. Technically, it is the message queues that are subscribed to the SNS topic, and they are each configured as an event source for their respective lambda functions so that when a queue receives a message, it automatically triggers the lambda function to process the message. Amazon SQS abstracts away the polling, reading and removal of messages which happen in the background.

4.4.2 Unenroll Member

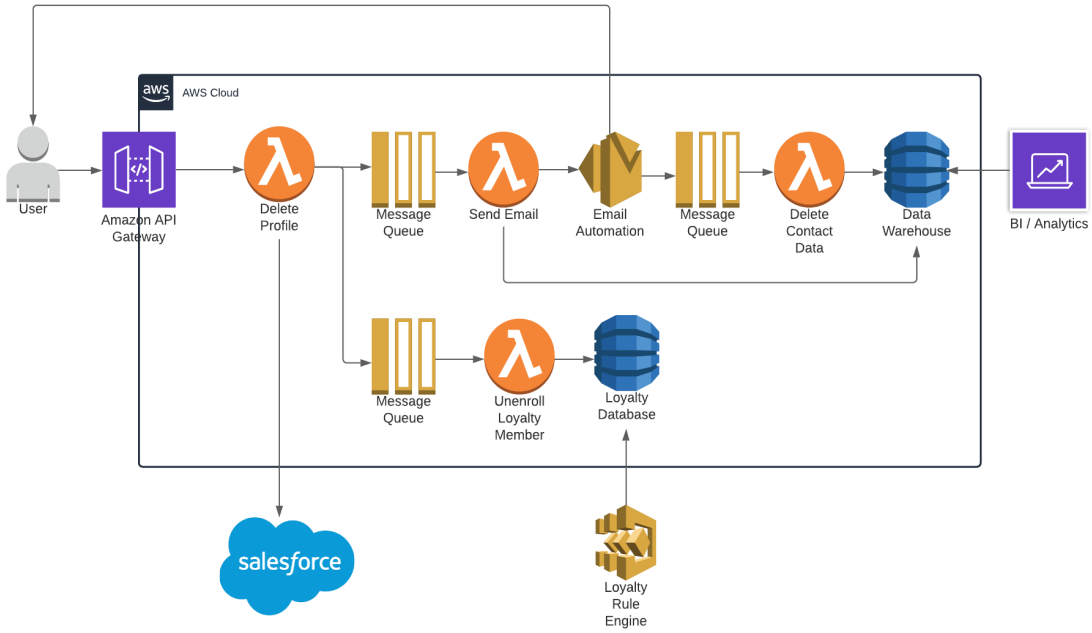

Figure 26. Architecture of integration flow for user deregistration

The member unenrollment process begins with the same *fanout* messaging pattern as the member enrollment process, but one of the two branches of the fanout tree uses a *Pipes and Filters* messaging pattern. The *Pipes and Filters* pattern breaks down a large processing task into a sequence of smaller, independent processing steps. The large processing task in this case can be summarized as “Send automated ‘goodbye’ email containing Contact data fetched from the Data Warehouse and then delete Contact record from data warehouse.” The two smaller processing steps that this processing task is broken down to are:

1. Send “goodbye” email using Contact data fetched from the Data Warehouse
2. Delete Contact record from the Data Warehouse, implemented by the *Send Email* and *Delete Contact Data* lambda functions respectively.

In the language of *Pipes and Filters*, these two lambda functions are the *filters* which are connected in a sequence by *pipes* consisting of SNS-SQS messaging channels. The key term here is *sequence* - in some cases, processing steps cannot be executed in parallel the way they are in *Enroll Member* because there are interdependencies between the processing steps. In this case, the interdependency is between the *Send Email* step and *Delete Contact Data* step, since they make use of the same underlying database. If *Send Email* and *Delete Contact Data* are executed in parallel, there is a risk of encountering a race condition in which the Contact record has already been deleted from the Data Warehouse at the moment when the *Send Email* tries to fetch the Contact record.

4.5 Systems Engineering

In addition to providing the functionality of application integration, the solution must also provide several key features of supporting technical infrastructure that are expected from a production EAI implementation. These features can be described broadly as systems engineering features, and fall under the categories of security, fault tolerance, and observability.

Security features protect the software system from vulnerabilities. In the context of EAI, these vulnerabilities can occur wherever the system is accessed either externally via the internet or internally by users within an organization. Thus, our EAI implementation must offer three main features to mitigate security vulnerabilities. It must (1) encrypt all communications that take place over the internet using an encryption protocol such as TLS; (2) enforce access control (authentication and authorization) for all public API endpoints; (3) govern access rights for deploying and modifying the EAI implementation to users within an organization.

Fault tolerance refers to how well a system continues to operate in the event of failure or faults within one or more of its components. We expect the EAI implementation to be fault-tolerant so that it can continue operating normally even when some part of the system fails. For example, if the integration component to one application throws an exception due to malformed data, then a fault tolerant EAI should ensure that (1) the scope of affected components within the system should be as limited as possible and (2) the affected components should recover gracefully from the exception and continue functioning normally preferably with little intervention.

Observability describes how well a system's internal states can be known by its external outputs, and involves three main features: logging, monitoring and alarms. An EAI system that has good observability should allow administrators to easily access logs collected from all components within the system so that process instances can be easily traced during debugging. It should also provide a way to continuously monitor the health of the system through statistics, data visualizations and synthetic monitoring (i.e., periodically executing test scripts on the production system which emulate transactions). Lastly, it should provide a way to alert administrators of failure by sending alarms that are triggered whenever a failure is detected through monitoring tools.

5 Case Implementation

In this section, I will go through the details of implementing the serverless EAI architecture. I first describe the technologies used in this implementation, before proceeding to explain them in the context of this implementation by going through the project structure, infrastructure-as-code, deployment process, back-office integrations, and messaging integrations.

5.1 Technologies and Structure

5.1.1 Programming Languages

As we have already seen, AWS Lambda serves as the compute service for executing workloads. It supports several different languages in its runtime environment including Node.js, Python, Ruby, Java, Go and .NET. For the Lambda functions in this implementation, I use the Node.js runtime environment which executes JavaScript code.

TypeScript is used during the deployment process and compiles into JavaScript at deployment time. Using TypeScript rather than JavaScript makes the code more readable and allows us to take advantage of object-oriented features. All libraries and dependencies are managed using the package manager yarn and are defined in the package.json file. Additionally, I use the Webpack module bundler which helps minimize the size of deployment packages and optimizes the performance of the lambda functions.

5.1.2 Amazon Web Services

In addition to Lambda functions, however, a number of AWS services are used in the implementation, some of which we have already mentioned in the architecture design such as SNS and SQS. All together, they are listed as follows:

- Amazon API Gateway
- Amazon CloudWatch
- AWS Certificate Manager
- AWS CloudFormation
- AWS Cloud Development Kit (CDK)
- Amazon DynamoDB
- AWS Identity and Access Management (IAM)
- AWS Lambda
- Amazon Simple Email Service (SES)
- Amazon Simple Notification Service (SNS)
- Amazon Simple Queue Service (SQS)
- Amazon S3
- AWS Systems Manager

While many of these services can be provisioned and configured manually through the web interface or using the AWS Command Line Interface (CLI), there are several Infrastructure-as-Code tools and SDKs which make it possible to provision AWS resources either programmatically or by using templates. CloudFormation is a native tool offered by AWS that can be used to provision and manage a set of related resources through user-defined models written as JSON or YAML templates.

CloudFormation templates are extremely flexible and powerful, but they can also quickly become verbose and difficult to manage when many services are involved. Hence, it is common to see the other infrastructure-as-code tools and frameworks such as Terraform, Serverless Framework or AWS Cloud Development Kit (CDK) being used as an alternative to CloudFormation. These tools provide a higher-level abstraction layer on top of the services, meaning that developers can specify composite services and allow the tools to automatically provision dependent resources, such as access policies, behind the scenes. While Terraform deploys templates using the AWS SDKs, Serverless Framework and CDK are built on top of CloudFormation and work by synthesizing the higher-level abstractions into “low-level” CloudFormation JSON templates before deploying them to the cloud.

Serverless Framework uses YAML templates to model the infrastructure resources. As its name suggests, Serverless Framework is designed specifically for serverless architectures and therefore offers a limited number of AWS services - these include the most used serverless services such as AWS Lambda and DynamoDB. Amazon CDK, on the other hand, is a fully-fledged SDK in which infrastructure resources can be modelled using a programming language, and is currently available in JavaScript / TypeScript, Python, Java, and .NET.

In this implementation project, I use CDK in TypeScript as the tool for provisioning infrastructure as code. This allows the implementation to use a single programming language for both infrastructure provisioning as well as the integration runtime, minimizing the number of development tools required. For instance, the yarn package manager is used to manage dependencies for both the lambda function code as well as the CDK infrastructure code, and a JavaScript framework such Jest can be used for unit testing both the infrastructure code as well as the lambda function code.

5.1.3 Project Structure

The project is divided into three main directories - /src contains the code used by the lambda functions, /stacks contains the code for CDK stacks which provision infrastructure

resources, and /cdk-app contains the code which defines the deployment and serves as the orchestrator for these CDK stacks. The /src/lambda subdirectory contains the Lambda function handlers, and the /src/lib subdirectory contains re-usable components such as the HTTP client class for accessing the Salesforce REST API.

The /stacks directory contains CDK stacks which represent the logical models for provisioning infrastructure in the form of TypeScript code. I have grouped these stacks based on the types of infrastructure they provision. For example, the /stacks/database subdirectory contains a stack that provisions a shared database, while the /stacks/messaging subdirectory contains several stacks that provision infrastructure such SNS topics and SQS queues.

```
├── cdk-app
│   └── cdk-app.ts
├── src
│   ├── lambda
│   │   ├── messaging
│   │   │   ├── delete_contact_sqs.ts
│   │   │   ├── ingest_contact_sqs.ts
│   │   │   ├── loyalty_enroll_sqs.ts
│   │   │   ├── loyalty_unenroll_sqs.ts
│   │   │   ├── publish_sns.ts
│   │   │   └── send_email_sqs.ts
│   │   └── salesforce
│   │       ├── create
│   │       │   ├── create_account.ts
│   │       │   └── create_contact.ts
│   │       ├── delete
│   │       │   ├── delete_account.ts
│   │       │   └── delete_contact.ts
│   │       ├── get
│   │       │   ├── get_account.ts
│   │       │   └── get_contact.ts
│   │       └── update
│   │           ├── update_account.ts
│   │           └── update_contact.ts
│   └── lib
│       ├── salesforce_client.ts
│       └── ssm_service.ts
├── stacks
│   ├── database
│   │   └── database-stack.ts
│   ├── domain-stack.ts
│   ├── messaging
│   │   ├── deletion-stack.ts
```

```
|   |   ├── email-stack.ts
|   |   ├── ingestion-stack.ts
|   |   ├── loyalty-stack.ts
|   |   └── messaging-stack.ts
|   ├── rest
|   |   ├── rest-stack.ts
|   |   ├── salesforce-account-proxy.ts
|   |   ├── salesforce-contact-proxy.ts
|   |   └── salesforce-proxy.ts
|   └── shared
|       ├── lambda-construct.ts
|       └── utils.ts
```

5.1.4 Infrastructure-as-Code

Using CDK, infrastructure can be provisioned programmatically as CloudFormation stacks using TypeScript. This is a major advantage that CDK has over some other infrastructure-as-code frameworks, which rely on static templates. These TypeScript stacks are then synthesized into CloudFormation templates and deployed to AWS using a command-line interface. The CDK app serves as a deployment scope for the different stacks and also provides a way to orchestrate the interconnections between the stacks.

In cdk-app.ts (Appendix 1), we define a CDK app as a deployment scope by simply instantiating it from the CDK App class:

```
import * as cdk from '@aws-cdk/core';
const app = new cdk.App();
```

The stacks which define the infrastructure models can then be added to the CDK app's scope to be included in the deployment. For example, to deploy the messaging stack containing contains the SNS topics, we simply instantiate the MessagingStack class that we have defined in /stacks/database/database-stack.ts:

```
const messagingStack = new MessagingStack(app, `eai-messaging`);
```

The MessagingStack, meanwhile, is defined by the following class which extends the CDK Stack class. It creates all three topics that are used by the messaging integrations:

```
import * as cdk from '@aws-cdk/core';
import * as sns from '@aws-cdk/aws-sns';
```

```
export class MessagingStack extends cdk.Stack {
 public createContactTopic: sns.Topic;
 public deleteContactTopic: sns.Topic;
 public unenrollMemberTopic: sns.Topic;

 constructor(scope: cdk.Construct, id: string) {
   super(scope, id);
   this.createContactTopic = new sns.Topic(this, `${id}-sns-createContact`, {
     displayName: 'Contact subscription topic'
   });
   this.deleteContactTopic = new sns.Topic(this, `${id}-sns-deleteContact`, {
     displayName: 'Delete contact subscription topic'
   });
   this.unenrollMemberTopic = new sns.Topic(this, `${id}-sns-unenrollContact`, {
     displayName: 'Unenroll member subscription topic'
   });
 }
}
```

The RestStack class (Appendix 2) contains the engagement layer lambda functions which publish messages to the SNS topics defined in the MessagingStack. For a Lambda function to be able to publish a message to an SNS topic, it needs to have (i) permission to access and publish a message to the topic and (ii) an address or identifier with which to reference the topic.

In CloudFormation, access permissions are usually managed by provisioning an IAM policy that grants permissions to a resource, using the resource's Amazon Resource Name (ARN) as a unique identifier. Since we are using CDK, however, we can make use of object-oriented practices to establish permissions programmatically. The CDK construct for an SNS topic has a function grantPublish to which a lambda function construct can be passed as an argument. Thus, we pass the CreateContact lambda construct as an argument to the SNS topic's grantPublish function to programmatically define the trust policy. At deployment time, CDK synthesizes the programmatically provisioned trust policy into a CloudFormation template that provisions an IAM policy granting the trust relationship.

```
// Api gateway serving as HTTP exposure gateway to applications
const restStack = new RestStack(app, `eai-rest`);

// Messaging stack contains topics used for communicating between applications
const messagingStack = new MessagingStack(app, `eai-messaging`);
```

```
messagingStack.createContactTopic.grantPublish(restStack.salesforce.contact.createContact);
```

For the lambda function to reference the SNS topic at runtime, it also needs to be able to access its ARN in the runtime environment. This can be accomplished by configuring a Lambda environment variable which becomes available as a global variable in the Node runtime. CDK Lambda constructs have an addEnvironment function which can be used for adding runtime environment variables.

```
restStack.salesforce.contact.createContact.addEnvironment('CREATE_CONTACT_TOPIC_ARN',
messagingStack.createContactTopic.topicArn)
```

As illustrated previously in the architecture, a message that is sent to an SNS topic is consumed by an SQS queue. This requires configuring the SQS queue to be added as a subscriber to the topic, and can be done in an object-oriented way using CDK. For the “send email” process in the Enroll Member flow, for example, we can instantiate the EmailStack, which contains an SQS queue and a Lambda function that sends emails, and configure the Create Contact SNS topic to add the queue as a subscriber.

```
// Marketing automation application
const emailStack = new EmailStack(app, `${APP_NAME}-email`);
messagingStack.createContactTopic.addSubscription(new
subs.SqsSubscription(emailStack.queue));
```

Within the EmailStack, the SQS queue is configured as an event source for the email sender lambda function, meaning that once a message is consumed from the SNS topic by the queue it is forwarded as an event to the email sender Lambda. We will examine how this is done from the Lambda functions shortly.

5.1.5 Deployment Process

As deploying AWS CDK apps into an AWS environment requires provisioning resources such as Amazon S3 buckets for storing deployment artifacts and IAM roles, the AWS environment needs to be “bootstrapped” when using CDK for the first time. After the AWS environment has been bootstrapped, the CDK command-line interface can be used to synthesize and deploy the CDK app and associated stacks.

```
cdk synth eai-rest
cdk deploy eai-rest
```

This deployment command is used to deploy the resources for the first time, as well as for any subsequent updates. When deploying a stack, CDK can diff the synthesized CloudFormation templates to determine whether anything has been modified. If there are no changes detected, CDK skips the deployment.

The serverless architecture in our implementation contains a total of eight different stacks that are defined using TypeScript in the cdk-app.ts. It would therefore be cumbersome if we must manually specify and deploy each stack independently. Luckily, CDK can infer most dependencies between stacks that are included in the CDK app scope. For example, since the MessagingStack is a dependency of RestStack, we only need to explicitly synthesize and deploy the RestStack using the above command, and CDK will automatically deploy the MessagingStack as well.

However, sometimes there are no programmatically defined dependency relationships. This is the case with the DomainStack which deploys the custom domain referenced statically in the RestStack's API Gateway. If we want CDK to nonetheless check the DomainStack for changes and deploy it along with the RestStack, we must manually specify it as a dependency like so:

```
const domainStack = new DomainStack(app, `eai-domain`);
restStack.addDependency(domainStack);
```

Combining the CDK commands into a deployment bash script that is executed by a Node npm script in package.json, we can then use a one-line deployment command to initiate the deployment. This command can be run from either the developer's machine or from a CI/CD environment:

```
yarn deploy
```

5.2 Architecture Implementation

5.2.1 Exposure Gateway

In the architecture design, we present an exposure gateway which allows services such as a web app backend to invoke our APIs from the internet. This is implemented using Amazon API Gateway and provisioned from the RestStack by the createApi function.

```
createApi(): apigateway.RestApi {
```

```
    return new apigateway.RestApi(this, `${this.id}-api`, {
      apiKeySourceType: apigateway.ApiKeySourceType.HEADER,
      cloudWatchRole: false,
      deploy: true,
      endpointConfiguration: {
        types: [apigateway.EndpointType.EDGE],
      },
    });
  }
```

As the reader might recall from the architecture design, we need to expose the internet-accessible APIs on the path endpoint https://eai.ceora.app/salesforce/contact. This is done by adding child resources to the API Gateway's root API construct.

```
const salesforce = this.api.root.addResource('salesforce');
const contact = salesforce.addResource('contact')
```

To configure burst and rate limits for the API gateway's usage, a usage plan is created for the API gateway as follows

```
createUsagePlan(api: apigateway.RestApi): apigateway.UsagePlan {
    const usagePlan = api.addUsagePlan(`${this.id}-usage-plan`, {
      throttle: {
        burstLimit: 50,
        rateLimit: 20,
      },
    });
    usagePlan.addApiStage({
      stage: api.deploymentStage,
    });
    return usagePlan;
  }
```

Finally, to make sure that the lambda functions are protected by some form of authentication, API keys are created and added to the usage plan. Here, we see an example of programmatic infrastructure provisioning in action using a for-loop to create API keys from a list.

```
createApiKeys(): void {
    const apiUsers = ['internal'];
    for (const user of apiUsers) {
```

```
    const apiKey = new apigateway.ApiKey(this, `${this.id}-api-key-${user}`, {
      apiKeyName: `${this.id}-${user}`,
      resources: [this.api],
    });
    this.usagePlan.addApiKey(apiKey);
  }
}
```

5.2.2 Engagement Layer

The engagement layer consists of the Create Profile and Delete Profile lambda functions which are accessible from the internet via the exposure gateway. These are also provisioned using the RestStack using the AWS Lambda SDK, which takes parameters such as the Lambda runtime, memory size, timeout duration, log retention.

```
this.lambda = new lambda.Function(this, id, {
  functionName: id,
  runtime: lambda.Runtime.NODEJS_12_X,
  memorySize: 128,
  timeout: cdk.Duration.seconds(12),
  logRetention: 1,
  code: lambda.Code.fromAsset('../../dist/bundle'),
  handler: 'create_contact.handler',
});
```

CDK packages the code assets from the specified asset directory and deploys it to the Lambda function. In our case, this asset directory contains the Webpack bundles compiled from our TypeScript code and JavaScript dependencies. The handler property specifies the file and function that contains the lambda logic to be executed by the Lambda handler. This pattern of defining a Lambda function in CDK is used by all the other infrastructure stacks in the implementation, including the SQS handler Lambda functions.

5.2.3 Back-Office Integrations

The back-office applications in this scenario are Salesforce, Loyalty System, Email Automation, and Data Warehouse. Except for Salesforce, all the applications are stubbed - however, we can nevertheless use Salesforce as an example to illustrate how the serverless integration runtime communicates with the back-office applications.

Salesforce offers a powerful REST API which can be used for machine-to-machine communication from the lambda functions. After configuring Salesforce to expose its REST API by creating a Connected App and setting up authorization using OAuth 2.0, we can now create, read, update, and delete data to and from Salesforce by sending HTTP requests to its REST API.

Authentication to Salesforce requires several secret values that should not be stored unencrypted in the codebase. One way to handle this problem would be to store the secrets in an encrypted file that is decrypted at deployment time and made accessible as environment variables in the application runtime. A preferable solution, and the one I have chosen, is to make use of the cloud provider's secrets management service that can fetch and decrypt secrets at runtime. As part of its Systems Manager service, AWS offers a feature called Systems Manager Parameters (SSM) which can be invoked through an SDK by lambda functions at runtime.

From the AWS browser interface, I created a JSON object in SSM called "salesforce_secrets" which contains all the parameters needed for authorization to Salesforce. This parameter can then be fetched in the integration runtime using the AWS SDK as follows:

```
const parameter = await ssm.getParameter({ Name: 'salesforce_secrets', WithDecryption: true
}).promise();
```

With the authentication secrets in place, we now need to develop the components for making HTTP requests to the Salesforce API. There are several open-source libraries that provide a wrapper around the native Salesforce API which make it easier to use. I chose jsforce as it is one of the most popular ones with good documentation. The two customer flows in this scenario - enroll member and unenroll member - are implemented as separate lambda functions, and both communicate with Salesforce. To make this functionality reusable, therefore, I created a SalesforceClient (Appendix 3) which can be instantiated by any lambda function to communicate with Salesforce.

Using these tools, initializing a client for the Salesforce API is as simple as fetching and parsing the authentication secrets from SSM, instantiating the SalesforceClient by passing in the authentication secrets, and then making an HTTP request to authenticate and login to Salesforce.

```
const client: SalesforceClient = new SalesforceClient(options);
```

```
await client.login();
```

With the SalesforceClient, we can perform a wide range of actions that are available the Salesforce REST API using only a few lines of code. For example, creating a new Contact record based on a JSON request body is a one-liner as follows:

```
const contact = await client.sobject('Contact').create(contactCreateRequest)
```

5.2.4 Messaging Integrations

Having gone through how the engagement layer processes a request, and how this request synchronously integrates with the back-office application Salesforce via remote procedure invocation of its REST API, we can now proceed to look at the integration layer's messaging patterns.

As we saw already, the messaging infrastructure is modeled and deployed using CDK, and the CreateProfile lambda function has been granted permission to publish a message to the CreateContact SNS topic. The unique identifier for the SNS topic has also been added as an environment variable *CREATE_CONTACT_TOPIC_ARN*, which can be accessed in the Lambda Node runtime. We therefore only need to fetch the topic ARN and use the SNS SDK to publish the message.

In Enroll Member Flow's Create Profile lambda, for example, we publish a successfully created contact record as a JSON-encoded message to the Create Contact topic as the first part of a fanout integration pattern.

```
const topicArn = process.env.CREATE_CONTACT_TOPIC_ARN;
const sns = new SNSClient({ region: "eu-central-1" });
await sns.send(
 new PublishCommand({
   Message: JSON.stringify(contact),
   TopicArn: topicArn,
 })
);
```

As shown, the SNS SDK makes it very simple to publish a message with just a few lines of code. This method of forwarding a Contact record is used in both the Enroll Member flow's fanout integration pattern as well as the Unenroll Member flow's pipes-and-filters integration pattern.

After the message is published to the SNS topic, it is immediately consumed by the SQS queues that are subscribed to it. In the Enroll Member flow, the message is consumed by three queues: the email automation system queue, loyalty system queue and the data warehouse queue. Each queue in turn has a lambda handler subscribed to it that listens for events. When the queue consumes the SNS topic's message, it emits an SQS event containing the message's contents which is subsequently consumed by the lambda handler. The lambda handler can then extract, parse, and process the message by executing some business logic.

The Send Email lambda handler, for example, is used in both the Enroll Member as well as the Unenroll Member flows. It has one Send Email queue that is subscribed to two SNS topics - the Create Contact topic and Unenroll Member topic - meaning that it consumes messages from both topics. The Send Email lambda listens to SQS events emitted from the Send Email queue and processes incoming events. This is implemented with just a few lines of code like so:

```
export const handler: SQSHandler = async (event: SQSEvent, _context: Context) => {
 try {
   for (const record of event.Records) {
     const notification = JSON.parse(record.body);
     const message = JSON.parse(notification.Message);
     await sendEmail(message);
     if (message.unenroll) {
       const snsTopic = process.env.UNENROLL_MEMBER_TOPIC_ARN;
       await publishMessage(notification.Message, snsTopic!);
     }
   }
 } catch (error) {
   console.log(error);
 }
};
```

With the SQS lambda handler, we can differentiate two use cases which have slightly different processes. In the Enroll Member flow, the lambda only sends a welcome email to the new member and the process ends - it serves as the message consumer in the fanout integration pattern. In the Unenroll Member flow, the lambda sends a goodbye email and then further publishes a message to another SNS topic, which notifies the Delete Contact lambda to initiate the member un-enrollment and cleanup process. This second flow uses the pipes-and-filters pattern in which the SQS lambda handler serves as the filter and subsequently pipes the message downstream to another filter for further processing.

5.3 Systems Engineering

As discussed in the case design, the serverless EAI implementation needs to provide key systems engineering features which ensure the health and security of its computing infrastructure. These, we saw, are features related to security, fault tolerance, and observability. Owing to the managed services provided by the AWS cloud environment, all the heavy lifting required in systems engineering is handled out-of-the-box in a serverless implementation. Rather than designing and building systems engineering components myself, therefore, I have simply made use of the managed services offered by AWS with minimal configuration or customization.

5.3.1 Security

Amazon API Gateway is the only part of the architecture exposed to the internet and by default it uses HTTPS to enforce TLS encryption for all connections. When deploying a Lambda function, Amazon API Gateway automatically assigns an internal domain to the API that uses the Amazon API Gateway certificate. For this implementation, I used a custom domain name – ceora.app – which is managed by the Google DNS service, and AWS Certificate Manager was used to issue a certificate for it. As a result, the system securely exposes public API endpoints such as https://eai.ceora.app/salesforce/contact/ via TLS encryption.

Access to the EAI's API endpoints via internet traffic is authenticated using API keys, which are configured on Amazon API Gateway. The API endpoints in the implementation are intended for server-to-server use, meaning that they are invoked by other servers such as a web application backend for example1. To emulate these servers, cURL and Postman were used for testing HTTP requests to the API endpoints. Indeed, more advanced protocols such as OAuth or JWT could have been used to provide authorization capabilities – however, such authorization features are not needed at this layer. Presumably authorization would be handled further upstream as part of a larger authentication service, from which the EAI stack's APIs are invoked.

AWS Identity and Access Management (IAM) is used in this implementation to provide access control to the AWS environment for users within the organization. Access to the AWS environment requires user credentials which grant both console access through the web browser as well as programmatic access through the CLI. As this implementation involved only one developer, fine-grained access control using IAM was not used. All deployment activities were executed by my AWS administrator user account, primarily using the AWS CLI with configured access keys. However, it would be trivial to provision IAM user credentials with fine-grained security controls if more developers were to be

onboarded to the project. Using the IAM console, an AWS administrator can configure access rights to individual resources either by creating access policies for an individual user, or for a group or role to which the user can then be assigned.

The fine-grained capabilities offered by IAM are nevertheless used for server-to-server access control. As shown in the previous section on Infrastructure-as-Code and the CDK application (Appendix 1), access between all deployed components is restricted by default and need to be explicitly specified by defining IAM access policies. Using this approach limits the exposure of security vulnerabilities within the AWS environment – for example, if a resource from an unrelated stack is compromised within the same AWS account, it cannot infiltrate the resources in the EAI stack because access is restricted by default. A more robust way of implementing cloud security at the enterprise level would be to sandbox services into separate AWS accounts – such as having a separate account for the EAI implementation - or even to sandbox each of the EAI's deployment environments into separate AWS accounts – such as development, UAT and production. Cross-account access, which are needed for scenarios such as continuous integration, can then be configured using IAM roles. This is however a vast topic that ventures far beyond the scope of this research - it suffices to say for our purposes that the AWS IAM service used in the implementation provides powerful fine-grained access control that can be highly customized for internal users and services.

5.3.2 Fault Tolerance

Serverless computing has built-in fault tolerance through several of its architectural and infrastructural features. First, because workloads are logically spread across multiple different lambda functions, failure within one lambda function can be relatively contained so that it does not bring down the entire system. For example, if the Send Email lambda function throws an unhandled exception due to a logical error, other lambda functions can continue to operate without interruption. In comparison, an uncaught exception in a monolithic application may very well crash the entire system.

Second, lambda functions maintain compute capacity across multiple Availability Zones – each representing a group of data centers – which effectively ensures that the function is available even if an individual machine or data center facility fails. Having distributed computing capacity allows the system to be resilient against hardware failure, and this is particularly the case for serverless workloads as rebalancing and other server maintenance tasks are completely automated behind the scenes. Lambda functions thus provide fault tolerance against hardware failure with minimal operational input needed.

Third, the decentralized messaging integration patterns used in this EAI architecture are designed fundamentally with fault tolerance in mind. Using an SQS queue as a buffer for asynchronous message processing means that a lambda message consumer does not need to be available when a message is published. Thus, in the event where the lambda has failed and is unavailable, the message remains on the queue and is read from the queue whenever the lambda becomes available again. SQS keeps processing messages until the expiration of the retention period.

Fault tolerance is one of the key technological advantages that the horizontal network design of serverless computing has over vertical hierarchical design. A monolithic integration runtime hosted on a single server presents a single point of failure, while a distributed infrastructure like the one implemented in this serverless computing allows the system to be resilient and highly available even when components fail.

5.3.3 Observability

Logging, monitoring, and alarms are all provided out-of-the-box through Amazon CloudWatch when deploying lambda functions. Amazon CloudWatch is AWS's monitoring and observability service that collects monitoring and operational data from logs, metrics, and events, and can be used to set alarms, trigger automated actions, visualize logs, and troubleshoot issues. When deploying a lambda function using CDK, a CloudWatch log group is automatically created which collects logs emitted in the lambda runtime. When troubleshooting issues, it is straightforward to access and search through the logs in the CloudWatch log groups.

For more advanced log searches, Amazon CloudWatch Log Insights provides a console with a purpose-built query language that can be used to search across multiple log groups. This is particularly useful for tracing process instances that span multiple lambda functions within the integration stack. A simple way of implementing tracing is to generate a unique request ID for each process instance and include it as a request header at each processing stage. When troubleshooting a process instance, we can then easily use CloudWatch Log Insights to query across multiple log groups using the unique request ID as a query parameter and thereby trace the process instance across multiple lambda functions.

Due to the stateless and autoscaling features of lambda functions, minimal infrastructure metrics are needed for this implementation. However, one useful metric is the API response times for integrations to back-office applications. For a production scenario where response times for the Salesforce API need to be monitored, one can add a

responseTime variable to the Salesforce client which measures how long it takes to receive a response from the Salesforce API after a request has been made. This *responseTime* value can then be emitted as a log within the lambda functions, which can then be queried using CloudWatch Log Insights. This data can then be aggregated into metrics using functions provided by CloudWatch's query syntax. These metrics, in turn, can serve as the basis for data visualization dashboards and alarms for use by an operations team.

6 Evaluation

6.1 Methodology and Limitations

Having gone through the implementation artifact, we can now evaluate its efficacy in EAI against the abstract model established from our theoretical framework. In Chapter 2, we defined an effective EAI to be one that *economizes on the production, coordination, and vulnerability costs of organization, while minimizing negative externalities incurred from search, decision-making, and enforcement*. Our model further predicts that serverless computing should perform well based on these criteria relative to alternative integration architectures. Let's therefore evaluate how well our implementation artifact has performed in terms of transaction costs, efficiency, and flexibility, and explore any unusual additional findings.

It is important to point out some of the limitations of this evaluation. Any assessment of the artifact is based primarily on my qualitative observation, and therefore it admittedly does not have a high standard of rigor. Furthermore, since we have not set up any metrics, control group or indeed any alternative integration architecture with which the artifact can be compared against, it is impossible to quantitatively verify any evaluative statements. However, this is not the main purpose of our evaluation. This being a design science research, our methods and subsequent results are exploratory. Our abstract model presents a hypothesis about serverless computing in EAI, and the implementation artifact can neither confirm nor reject this hypothesis. Evaluating the implementation artifact merely allows us to qualitatively verify the patterns presented in the abstract model and account for any unusual or surprising findings. More rigorous quantitative analyses in further research can thus be pursued based on these exploratory findings.

To evaluate the implementation artifact, we must infer its transaction, production, coordination, and vulnerability costs from empirically observed technological and organizational factors. Costs from organizational factors (such as coordination costs) are reflected in the time and labor required for task assignment, and therefore the simplest way to empirically observe organizational factors in this research would be to measure the time taken to implement a solution. In an organization, time taken to implement a solution translates directly into labor costs, which often constitutes the largest expense. However, given the limitation of this research (namely that we have no control group, and the implementation was completed by only one developer), measuring the overall time it takes to implement this case architecture would not yield meaningful results. We can nevertheless determine the relative time taken by certain activities within the implementation process – expressed as a subjective measure of complexity – and use

that as an empirical observation. For instance, observing the time it takes to complete a task that typically requires organizational cooperation, such as developing a messaging channel between two applications, gives us an idea of the transaction and coordination costs of serverless when we compare it to alternative technologies. For simplicity's sake, therefore we will take cloud costs as a measurement of technological factors, and the relative complexity (time taken) as a measurement of organizational factors.

6.2 Model Verification

6.2.1 Transaction Costs

Our abstract model suggests that serverless architecture is better suited than alternatives to economize the transaction costs of search, decision-making and enforcement due to its unique combination of hierarchical and network characteristics.

As a quick review, we saw that horizontal networks such as point-to-point topologies incur high costs from uncertainty, bounded rationality, information asymmetry and asset specificity when developing integrations, as each application must integrate individually with one another using market-like mechanisms. These costs are addressed by the emergence of hierarchical structures such as hub-and-spoke and SOA, but at the expense of production, coordination, and vulnerability costs. Decentralized microservices emerged to mitigate the production, coordination and vulnerability costs found in SOA architectures, but its horizontal network characteristics once again result in the kinds of transaction costs we found in point-to-point integrations - this time, as the negative externalities of flexibility. Administrative tools for operational tasks such as service discovery, deployment, security, audit logging, and monitoring must often be implemented in addition to the integrations themselves, which incur significant costs. Our abstract model posits that serverless should fare better at handling these negative externalities because its full application functionality provides many of the same benefits of a vertical hierarchy like an ESBs.

Indeed, my experience of implementing the serverless EAI architecture has absolutely confirmed this. Despite its distributed design, which is characteristic of horizontal networks, serverless still retains some of the hierarchical elements of ESBs in the form of out-of-the-box features, APIs, and SDKs that take advantage of economies of scale. Take logging for example. Logging with AWS CloudWatch came out-of-the-box with each of the deployed lambda functions, and retention times are easily configurable in CDK using one line of code. We can contrast this to the amount of work required to manually implement and maintain logging tools such as ElasticSearch and Kibana, which are commonly found

in containerized microservice architectures and can sometimes require dedicated projects or personnel to implement.

Security is another example where serverless architecture EAI performs with flying colors. As we saw when going through the integration to Salesforce, AWS offers out-of-the-box secrets management tools such as SSM that we can use to store authentication secrets, mitigating the security risks - as well as development costs - of encrypting and decrypting secrets in the codebase. We were also able to create API keys for our two internet-exposed lambda functions using CDK with just a few lines of code. Admittedly, basic authentication using API keys does not necessarily represent production-grade security, however this has been chosen to make the functions easily accessible for testing. Implementing more advanced security protocols such as signed requests with AWS Signature Version 4 is straightforward and does not require much more effort.

6.2.2 Production Costs

The fixed production costs of this serverless EAI architecture are extremely low. For the month of April 2021, during which the integration runtime was almost completely idle, the cost of hosting the entire architecture on AWS was $1.22. Furthermore, these costs were incurred from only one service, SQS, which needs to be constantly running to poll for incoming messages. All other services such as SNS, Lambda, S3 and CloudWatch - because they were not used - all remained within the limits of the free tier. The $1.22 monthly cost thus gives us a baseline of the fixed cost of running five SQS queues each configured with a long polling wait time of 20 seconds. Clearly, therefore, the fixed production costs of serverless without a doubt blow all alternatives out of the water when considering the costs of maintaining stateful solutions EC2 instances or, even worse, license fees of commercial ESBs.

To evaluate the overall production costs of serverless, however, we would need to account for its marginal production costs - and indeed, the pay-as-you-go pricing model of serverless is based almost entirely on marginal costs. This would require that we put a higher load on the serverless runtime, perhaps based on production environments, and observe how it compares to alternatives running the same load.

An important point, however, is that the production costs on serverless are determined not only by the amount of load, but by its pattern. Serverless is ideal for stateless services such as REST APIs, especially where load is unpredictable and characterized by alternating peaks and periods of idleness. Using serverless for services characterized by constant, predictable load, may in fact make it more expensive than using stateful

containerized microservices. For the purposes of my implementation however, the pay-as-you-go pricing model is perfect, as my use case is exactly one where load is characterized by long periods of idleness.

6.2.3 Coordination Costs

Our abstract model suggests that serverless should economize on coordination costs more effectively than alternatives, and I have observed this to be true while caveated by the limitations of my methodology. Coordination costs are the costs of processes such as task delegation and decision-making - within the software engineering teams, this translates to processes such as releasing software. We saw with ESBs that in vertical hierarchies, coordination costs are high since centralization results in the formation of a specialized ESB unit that effectively becomes a single point of failure and blocker.

By using infrastructure-as-code with CDK, we can define and deploy infrastructure components independently as modular units. Combined with the CLI, SDKs and APIs offered by AWS, we can easily setup scripts with which infrastructure can be independently deployed and torn down. For example, we can establish a data governance model that is well-defined yet highly flexible. By default, the serverless architecture assumes a database-per-service pattern in line with microservice principles, but we can set up access to the database stack so that it serves as a shared database integration between the data warehouse and the email automation system.

Admittedly, however, it is difficult to evaluate how well our serverless EAI architecture fares in terms of coordination costs since there is no one other than the author of this research who is building integrations towards it. Since this implementation has involved only one developer, it has for the most part followed a centralized deployment model - there is only one CDK app which deploys all the infrastructure stacks. This implementation has not been able to fully take advantage of the decentralized coordination features of serverless.

I suspect, however, that if several developers were to develop new integrations to this serverless runtime, we would find that they are able to work independently without many bottlenecks. One simple way that the decentralized coordination features of serverless architecture could have been tested is by implementing parts of the integration architecture in another programming language. For example, we could have implemented the Email Sender entirely in Python, using the Python CDK library and Lambda runtime. This would have provided a way to emulate a situation where two different teams must

develop integrations to the same runtime and would have been a good way to test how well serverless infrastructure handles the coordination costs involved.

6.2.4 Vulnerability Costs

Based on the features offered by AWS, I've observed that running serverless architecture in AWS is resilient against vulnerability costs within the cloud platform, but not necessarily beyond it. Recalling that vulnerability costs are the "unavoidable costs of a changed situation that are incurred before the organization can adapt to the new situation" such as technological shock, our abstract model suggests that horizontal networks in general are better equipped than vertical hierarchies at handling vulnerability costs. This is because the lower coordination costs found in horizontal networks also lowers the costs of learning, and therefore horizontal networks have a higher propensity to innovate.

This propensity to innovate is epitomized by microservice architectures, which have maximal freedom to experiment with different tools and technologies. Serverless computing also does a good job at being compatible with a wide range of programming languages and tools, especially when compared to the limitations imposed by alternatives such as ESBs. However, it nonetheless has a limited hand due to the limitations imposed by cloud infrastructure. For example, cold start times make it largely prohibitive to use JVM technologies in Lambda functions, meaning that serverless developers cannot take advantage of the latest developments in that world such as Kotlin.

Moreover, I found that the implementation needed at times to be vendor-specific to make the most out of the coordination advantages offered by serverless in AWS. From the perspective of transaction cost economics, this can be seen as a form of asset-specificity cost - i.e., the cost incurred to reallocate the asset for another use. For example, the lambda functions contain logic that needs to be developed specifically to handle AWS events such as those from API Gateway or SQS. While these pieces of vendor-specific code are not overly intrusive, they would at the minimum need to be refactored to be used in another cloud environment.

More problematic are the technologies that I have lauded in previous sections of this evaluation, namely CDK and the rich set of tools which allow serverless to take advantage of both vertical hierarchy as well as horizontal network characteristics. These technologies, particularly those that I have referred to as out-of-the-box administrative features, are crucial mechanisms for economizing transaction and coordination costs, and yet they are also the most vendor-specific. This vendor-specificity means that in the case

of vendor change, any investments made in AWS present not only non-transferrable sunk costs, but active vulnerability costs from refactoring and testing.

6.3 Summary of Findings

To summarize the findings of this research and evaluation, let's return to the three research questions which I set out to answer and see how well we have been able to answer them throughout this thesis.

RQ1: What are the main existing approaches to EAI and what are their main problems?

In Chapters 2 and 3, we found that main approaches to EAI can be described by a set of integration patterns and integration architectures. These integration patterns fall under four main types: file transfer, shared database, remote procedure invocation and messaging. Integration patterns are mixed and matched to form integration architectures, and we identified these to be point-to-point, hub-and-spoke, service-oriented architecture and microservices. Using the theoretical framework and abstract model developed in Chapter 2, we saw that these different EAI architectures can be categorized as either vertical hierarchies or horizontal networks and that the problems they encounter can be understood in terms of tradeoffs between transaction, production, coordination, and vulnerability costs.

RQ2: How can serverless be used in EAI?

In Chapter 3, we introduced serverless computing as a technology and examined its applications as an integration runtime for EAI. We then presented a design and implementation artifact of a serverless integration runtime using AWS in Chapters 4 and 5, using a hypothetical scenario in which several back-office enterprise applications must be integrated behind a customer-facing interface.

RQ3: How well does serverless solve the main problems of EAI?

Before designing and implementing the artifact, we evaluated the features of serverless computing in Chapter 3 using the abstract model of EAI and postulated that it should be an attractive approach to EAI due to its ability to economize on production costs, coordination costs, and vulnerability costs while being able to minimize the negative externalities resulting from decentralization. We then attempted to qualitatively verify this model by evaluating the artifact against it and found that our observations of the artifact are for the most part consistent with the model.

One interesting pattern we can infer from our implementation artifact is that the differences between microservice and serverless architecture can be interpreted as tradeoffs between transaction costs, coordination costs and vulnerability costs. Microservice architectures economize on vulnerability costs by avoiding vendor lock-in, at the expense of missing out on the rich set of vendor-specific features offered by cloud computing. Serverless architectures, on the other hand, make the most out of vendor-specific features which economize on coordination costs, at the expense of incurring vulnerability costs from asset-specificity.

7 Further Research

Several points for further research were identified throughout the evaluation process.

First, a load testing experiment can be conducted to examine the effect different load patterns have on the production costs of serverless computing. For example, we can create a test suite consisting of constant, step, and goal-based load patterns and record the costs incurred in AWS over a fixed time frame. We can then run the same test for an alternative integration runtime such as containerized microservices and quantitatively compare the results.

Second, as we noted earlier, the limitation of having only one developer meant that it was not possible to accurately observe the serverless EAI artifact's coordination costs. We would therefore benefit from a qualitative study in which several integration teams are observed developing integrations using serverless technologies. Documenting their ways of working and efficiency over a period could help us understand how coordination costs are handled by serverless architecture. A more ambitious experiment would be to observe many different teams from many different companies, control for serverless integration runtimes, and quantitatively measure team performance using metrics within a framework such as Scrum.

Given that this research is exploratory in nature, it has developed several abstractions regarding information systems and organizations which could be verified, refuted, or expanded upon. Three of these abstractions have been referenced throughout this thesis to describe and evaluate integration patterns and approaches. These are (i) the categorization of information systems as vertical hierarchy and horizontal network structures; (ii) the application of market mechanisms and transaction cost economics to the analysis of integration architectures; and (iii) the abstract model of EAI architecture as economizing on production, coordination, and vulnerability costs.

These abstractions make assumptions that have not been verified, and therefore they would certainly benefit from further research that either verify or refute the underlying assumptions. Alternatively, further exploratory research can be made to expand on the theoretical framework and identify what it has ignored, or account for any unusual findings.

References

Bechtel, W. 2020. Hierarchy and levels: analysing networks to study mechanisms in molecular biology. Philosophical Transactions of the Royal Society B, 375. URL: https://royalsocietypublishing.org/doi/10.1098/rstb.2019.0320. Accessed: 9 May 2021.

Charboneau, T. 15 June 2020. Takeaways From the 2020 Cloud Elements State of API Integration Report. Nordic APIs blog. URL: https://nordicapis.com/takeaways-from-the-2020-cloud-elements-state-of-api-integration-report/. Accessed: 4 May 2021.

Chaudhary, P., Hyde, M. & Rodger, J. 2017. Exploring the Benefits of an Agile Information System. Intelligent Information Management, 9, pp. 133-155.

Ciborra, C. 1983. Markets, Bureaucracies and Groups in the Information Society: An Institutional Appraisal of the Impacts of Information Technology. 10.1057/9780230250611_12. URL: https://www.sciencedirect.com/science/article/abs/pii/0167624583900240. Accessed: 4 May 2021.

Clark, K. May 2018. The fate of the ESB. From its origins in the SOA era to the challenges that inspired the search for a better approach. IBM. URL: https://developer.ibm.com/articles/cl-lightweight-integration-1/ Accessed: 4 May 2021.

Coase, R. H. 1937, The Nature of the Firm. Economica, 4, pp. 386-405. URL: https://doi.org/10.1111/j.1468-0335.1937.tb00002.x. Accessed: 4 May 2021.

Cordella, A. 2006. Transaction Costs and Information Systems. Journal of Information Technology, 21, pp. 195-202. URL: https://www.researchgate.net/publication/30522507_Transaction_Costs_and_Information_Systems, Accessed: 4 May 2021.

Corominas-Murtra, B., Goñi, J., Sole, R. & Rodríguez-Caso, C. 2013. On the origins of hierarchy in complex networks. Proceedings of the National Academy of Sciences of the United States of America, 110, 10.

Conway, Melvin E. April 1968. How do Committees Invent?. Datamation, 14, 5, pp. 28–31

Ferguson, N. 10 June 2014. Networks and Hierarchies. Niall Ferguson blog. URL: http://www.niallferguson.com/journalism/miscellany/networks-and-hierarchies. Accessed: 4 May 2021.

Ferguson, N. 2017. The Square and the Tower: Networks, Hierarchies and the Struggle for Global Power. Allen Lane. London.

Fong, J. 30 August 2018. Are Containers Replacing Virtual Machines? Docker blog. URL: https://www.docker.com/blog/containers-replacing-virtual-machines/ Accessed: 4 May 2021.

Gartner 2021. Application Integration. URL: https://www.gartner.com/en/information-technology/glossary/application-integration. Accessed: 4 May 2021.

Gilani, Aziz. March 2020. How the information system industry became enterprise software. TechCrunch. URL: https://techcrunch.com/2020/03/08/how-the-information-system-industry-became-enterprise-software/ Accessed: 4 May 2021.

Gulledge, Thomas. 2006. What is integration?. Industrial Management and Data Systems, 106, pp. 5-20.

Hevner, A. R., March, S. T., Park, J. & Ram, S. 2004. Design Science in Information Systems Research. MIS Quarterly, 28, pp. 75-106.

Hohpe, G., Woolf, Bobby. 2003. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. URL: https://www.enterpriseintegrationpatterns.com. Accessed: 4 May 2021.

HubSpot 2021. HubSpot Salesforce Integration. URL: https://www.hubspot.com/products/salesforce. Accessed: 4 May 2021.

Hunt, A. & Thomas, D. 2000. The pragmatic programmer: from journeyman to master. Addison-Wesley Longman Publishing Co., Inc., Boston. URL: https://biodieta.noblogs.org/files/2020/03/The-Pragmatic-Programmer_-your-journey-to-mastery-20th-Anniversary-Edition-2nd-Edition.pdf. Accessed: 11 May 2021.

IBM Cloud Education 2021. SOA (Service-Oriented Architecture). URL: https://www.ibm.com/cloud/learn/soa. Accessed: 4 May 2021.

Jung, D. & Lake, D. 2011. Markets, Hierarchies, and Networks: An Agent-Based Organizational Ecology. American Journal of Political Science, 55, pp. 972 - 990.

Kotter, John P. May 2011. Hierarchy and Network: Two Structures, One Organization. Harvard Business Review. URL: https://hbr.org/2011/05/two-structures-one-organizatio. Accessed: 5 May 2021.

Kökörčený, M. 2014. A new cost model for comparison of Point to Point and Enterprise Service Bus integration styles. 5th European Conference of Computer Science (ECCS '14), pp. 63-70. URL: http://www.wseas.us/e-library/conferences/2014/Geneva/ECCS/ECCS-10.pdf. Accessed: 4 May 2021.

Laudon, K. C., Laudon, J. P., & Brabston, M. E. 2005. Management information systems: Managing the digital firm. Prentice Hall. Toronto. URL: https://paginas.fe.up.pt/~acbrito/laudon/ch3/chpt3-2main.htm. Accessed: 4 May 2021.

MacCormack, A., Baldwin, C. & Rusnak, J. 2012. Exploring the Duality Between Product and Organizational Architectures: A Test of the ‘mirroring’ Hypothesis. Research Policy, 41, 8, pp. 1309–1324

Malone, T.W. & Smith, S. A. 1984. Tradeoffs in designing organizations: implications for new forms of human organizations and computer systems. URL: http://dspace.mit.edu/handle/1721.1/2075. Accessed: 4 May 2021.

Malone, T.W. 1985. Organizational Structure and Information Technology: Elements of a Formal Theory. URL: https://dspace.mit.edu/handle/1721.1/2123. Accessed: 4 May 2021.

Malone, T W. 1986. A Formal Model of Organizational Structure and Its Use in Predicting Effects of Information Technology. Sloan School of Management Working Paper, 1849, 86. URL: https://core.ac.uk/download/pdf/4379816.pdf. Accessed: 4 May 2021.

McKinsey 2020. Cloud 2.0: Serverless architecture and the next wave of enterprise offerings. URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/tech-forward/cloud-20-serverless-architecture-and-the-next-wave-of-enterprise-offerings. Accessed: 4 May 2021.

Nygard, M. 20 February 2009. Why Do Enterprise Applications Suck? Wide Awake Developers blog. URL: https://www.michaelnygard.com/blog/2009/02/why-do-enterprise-applications-suck. Accessed: 4 May 2021.

ONEiO 2020. Integration 2020. URL: https://campaign.oneio.cloud/integration-2020-report/. Accessed; 4 May 2021.

Qian, Y., Roland, G. & Xu, C. 1999. Coordinating Changes in M-form and U-form Organizations. William Davidson Institute at the University of Michigan, William Davidson Institute Working Papers Series. URL:

https://www.researchgate.net/publication/23724240_Coordinating_Changes_in_M-form_and_U-form_Organizations. Accessed: 4 May 2021.

Qian, Y., Roland, G., & Xu, C. 2006. Coordination and Experimentation in M-Form and U-Form Organizations. Journal of Political Economy, 114, 2, pp. 366-402

Richardson, C. 2020. What are microservices? URL: https://microservices.io/ Accessed: 9 May 2021.

Sapho 2016. Study: Employees Don't Use Enterprise Applications; Crave Personalized Experiences [Press Release]. URL: https://prnewswire.com/news-releases/study-employees-dont-use-enterprise-applications-crave-personalized-experiences-300317422.html. Accessed: 4 May 2021.

Tiernan, C. 24 November 2014. Why Point-to-Point Integrations Are Evil. Chris Tiernan blog. URL: https://www.christiernan.com/why-point-to-point-integrations-are-evil/. Accessed: 4 May 2021.

Tubaro, P. 7 April 2016. Hierarchy, market or network? The disruptive world of the digital platform. Data big and small blog. URL: https://databigandsmall.com/2016/04/07/hierarchy-market-or-network-the-disruptive-world-of-the-digital-platform/. Accessed: 4 May 2021.

Vaishnavi, V., Kuechler, W. & Petter, S. 2004. Design Science Research in Information Systems. URL: http://www.desrist.org/design-research-in-information-systems. Accessed: 4 May 2021.

Williams, A. August 2019. Enterprise software is hot — who would have thought?. TechCrunch. URL: https://techcrunch.com/2019/08/22/enterprise-software-is-hot-who-would-have-thought/. Accessed: 4 May 2021.

Williamson, O. 1991. Comparative Economic Organization: The Analysis of Discrete Structural Alternatives. Administrative Science Quarterly, 36, 2, pp. 269-296.

Young, S. 2013. Transaction Cost Economics. Encyclopedia of Corporate Social Responsibility. URL: https://doi.org/10.1007/978-3-642-28036-8_221. Accessed: 4 May 2021.

Appendices

Appendix 1. CDK App source code

```
#!/usr/bin/env node
import * as cdk from '@aws-cdk/core';
import * as subs from '@aws-cdk/aws-sns-subscriptions';
import { RestStack } from '../stacks/rest/rest-stack';
import { DatabaseStack } from '../stacks/database/database-stack';
import { MessagingStack } from '../stacks/messaging/messaging-stack';
import { EmailStack } from '../stacks/messaging/email-stack';
import { IngestionStack } from '../stacks/messaging/ingestion-stack';
import { LoyaltyStack } from '../stacks/messaging/loyalty-stack';
import { DomainStack } from '../stacks/domain-stack';
import { DeletionStack } from '../stacks/messaging/deletion-stack';

const APP_NAME = 'eai';
const app = new cdk.App();

// Custom domain name
const domainStack = new DomainStack(app, `${APP_NAME}-domain`);

// Stack for shared database integrations
const databaseStack = new DatabaseStack(app, `${APP_NAME}-database`);

// Api gateway serving as HTTP exposure gateway to applications
const restStack = new RestStack(app, `${APP_NAME}-rest`);
restStack.addDependency(domainStack);

// Messaging stack contains topics used for communicating between applications
const messagingStack = new MessagingStack(app, `${APP_NAME}-messaging`);
messagingStack.createContactTopic.grantPublish(restStack.salesforce.contact.createCon
tact);
messagingStack.deleteContactTopic.grantPublish(restStack.salesforce.contact.deleteCon
tact);
restStack.salesforce.contact.createContact.addEnvironment('CREATE_CONTACT_TOPIC_ARN',
messagingStack.createContactTopic.topicArn)
restStack.salesforce.contact.deleteContact.addEnvironment('DELETE_CONTACT_TOPIC_ARN',
messagingStack.deleteContactTopic.topicArn)

// Marketing automation application
const emailStack = new EmailStack(app, `${APP_NAME}-email`);
messagingStack.createContactTopic.addSubscription(new
subs.SqsSubscription(emailStack.queue));
databaseStack.contactsTable.grantReadWriteData(emailStack.emailSenderLambda);
emailStack.emailSenderLambda.addEnvironment('CONTACTS_TABLE',
databaseStack.contactsTable.tableName);
messagingStack.deleteContactTopic.addSubscription(new
subs.SqsSubscription(emailStack.queue));
messagingStack.unenrollMemberTopic.grantPublish(emailStack.emailSenderLambda);
emailStack.emailSenderLambda.addEnvironment('UNENROLL_MEMBER_TOPIC_ARN',
messagingStack.unenrollMemberTopic.topicArn);
restStack.addDependency(emailStack);

// Ingestion pipeline to store contacts to data warehouse
const ingestionStack = new IngestionStack(app, `${APP_NAME}-ingestion`);
messagingStack.createContactTopic.addSubscription(new
subs.SqsSubscription(ingestionStack.queue));
ingestionStack.contactIngestionLambda.addEnvironment('CONTACTS_TABLE',
databaseStack.contactsTable.tableName)
databaseStack.contactsTable.grantReadWriteData(ingestionStack.contactIngestionLambda)
;
restStack.addDependency(ingestionStack);

// Loyalty member services - enroll, process and uneroll member
const loyaltyStack = new LoyaltyStack(app, `${APP_NAME}-loyalty`);
```

```
messagingStack.createContactTopic.addSubscription(new
subs.SqsSubscription(loyaltyStack.enrollmentQueue));
messagingStack.deleteContactTopic.addSubscription(new
subs.SqsSubscription(loyaltyStack.unenrollmentQueue));
restStack.addDependency(loyaltyStack);

// Delete Contact
const deletionStack = new DeletionStack(app, `${APP_NAME}-deletion`);
messagingStack.unenrollMemberTopic.addSubscription(new
subs.SqsSubscription(deletionStack.queue));
deletionStack.deleteContactLambda.addEnvironment('CONTACTS_TABLE',
databaseStack.contactsTable.tableName)
databaseStack.contactsTable.grantReadWriteData(deletionStack.deleteContactLambda);
restStack.addDependency(deletionStack);
```

Appendix 2. RestStack source code

```
import * as cdk from '@aws-cdk/core';
import * as apigateway from '@aws-cdk/aws-apigateway';
import { loadConfig } from '../shared/utils';
import { SalesforceProxy } from './salesforce-proxy';

const config: object = loadConfig();
export class RestStack extends cdk.Stack {
 public id: string;
 public scope: cdk.Construct;
 public api: apigateway.RestApi;
 public usagePlan: apigateway.UsagePlan;
 public salesforce: SalesforceProxy;

 constructor(scope: cdk.Construct, id: string) {
   super(scope, id);
   this.id = id;
   this.scope = scope;
   this.api = this.createApi();

   const domainName = apigateway.DomainName.fromDomainNameAttributes(this, `${id}-
domain`, {
     domainName: config['domainName'],
     domainNameAliasHostedZoneId: '',
     domainNameAliasTarget: '',
   });
   new apigateway.BasePathMapping(this, `${id}-basePathMapping`, {
     basePath: '',
     domainName: domainName,
     restApi: this.api,
   });

   this.usagePlan = this.createUsagePlan(this.api);
   this.createApiKeys();

   this.salesforce = new SalesforceProxy(this, `${this.stackName}-salesforce`, {
     api: this.api,
   });
 }

 createApi(): apigateway.RestApi {
   return new apigateway.RestApi(this, `${this.id}-api`, {
     apiKeySourceType: apigateway.ApiKeySourceType.HEADER,
     cloudWatchRole: false,
     deploy: true,
     endpointConfiguration: {
       types: [apigateway.EndpointType.EDGE],
     },
   });
 }

 createUsagePlan(api: apigateway.RestApi): apigateway.UsagePlan {
   const usagePlan = api.addUsagePlan(`${this.id}-usage-plan`, {
     throttle: {
       burstLimit: 50,
       rateLimit: 20,
     },
   });
   usagePlan.addApiStage({
     stage: api.deploymentStage,
   });
   return usagePlan;
 }

 createApiKeys(): void {
```

```
    // Create API keys for each user
    const apiUsers = ['internal'];
    for (const user of apiUsers) {
      const apiKey = new apigateway.ApiKey(this, `${this.id}-api-key-${user}`, {
        apiKeyName: `${this.id}-${user}`,
        resources: [this.api],
      });
      this.usagePlan.addApiKey(apiKey);
    }
  }
}
```

Appendix 3. SalesforceClient source code

```
import { Connection, UserInfo } from 'jsforce';

export interface SalesforceOptions {
 endpoint: String;
 clientId: String;
 clientSecret: String;
 username: String;
 password: String;
 securityToken: String;
}

export class SalesforceClient extends Connection {
 private options: SalesforceOptions;

 constructor(options: SalesforceOptions) {
   super({
     oauth2: {
       clientId: `${options.clientId}`,
       clientSecret: `${options.clientSecret}`,
     },
   });
   this.options = options;
 }

 public async login(): Promise<UserInfo> {
   return await super.login(
     `${this.options.username}`,
     `${this.options.password}${this.options.securityToken}`
   );
 }
}
```