The effect of using APIs on Coordination during Software Development

Abstract

Coordination refers to the organization of the different actors of system so as to make them work in a harmonious manner towards a common goal. Software Engineering, like any complex collaborative process, requires a high degree of coordination. This has been well documented in Information Systems literature. However, the advent of new technologies and standards in the industry have pushed it towards adopting Application Programming Interfaces as a new form of coordination. This essay analyses the role of APIs in the coordination process using Malone & Crowston's Coordination Theory. The key actors, activities, goals and interdependencies are identifed and analysed. Key to this analysis was a deep rooted understaing of APIs as as IS artefact. Theoretical and practical implications of the same point towards an emerging form of coordination with its own unique challenges.

Introduction

Coordination and communication form essential parts of any collaborative process. The idea of coordination has received attention from various disciplines, with each contributing a new perspective to the general idea behind coordination. Even in Information Systems literature, the idea of coordination has received wide spread attention. Software Engineering specifically has been recognized as a highly complex process where communication and collaboration play key roles (Curtis et al. 1988).

In software engineering, an *interface* refers to the group of services provided by a piece of software while *implementation* refers to its actual code. The term, Application Programming Interface (API) refers to a formalized group of interfaces. Since its inception in 2000, this has become widely popular in the industry. APIs are essentially instantiations of the practice of information hiding in Software Engineering. For the purposes of this essay, a definition of APIs provided by des Rivières (2004) shall be used:

> "a well-defined interface that allows one software component to access programmatically another component and is normally supported by the constructs of programming languages."

There has been a recent industry trend of APIs becoming a key part of Software Engineering. This essay investigates the role played by APIs in facilitating coordination between software engineers. It recognizes the key actors that use APIs to coordinate and the various interdependencies associated with them. This essay also highlights the challenges of analysing APIs as traditional IS artefacts.

This essay first introduces the concept and literature around Application Programming Interfaces. Following this, a brief summary of the Software Development literature for the benefit of added context is presented. The section ends by summarizing the literature around coordination in the field of Information Systems. The next section builds a deeper

understanding of the theory behind Coordination and builds an appropriate framework for analysis. This is followed by introducing the Research Methodology used for this essay. In the Findings and Analysis section, key findings from the nine semi structured interviews that were conducted in person are presented. They are grouped into themes and analysed using the theoretical framework presented earlier. This is followed by a higher level overview of the entire Coordination process using two different theoretical lenses for a more complete understanding. The study ultimately concludes by summarizing its findings and presenting the potential limitations.

Critical Literature Review

Application Programming Interface

Application Programming Interface (API) is a set of end points providing tools for building an application or software. The term was introduced in Fielding's (2000) doctoral thesis where the architectural pattern of APIs was brought forth to demarcate the difference between the web and other middleware. Fielding pointed towards a library based API where a set of code entry points and parameters were defined by a programmer such that other programmers could use their code without knowing or worrying about its actual implementation.

APIs in essence are forms of software abstraction. Abstraction, in software engineering, is a construct to reduce the level of complexity by reducing information to essentials while ignoring background details (Brooks 1987). A complex system constitutes several levels of abstraction, wherein each layer encapsulates some function. The ability to reduce an object or a collection of objects to one abstract feature is crucial in software engineering (Hazzan & Tomayko 2005). Another important facet of abstraction is deciding the level of abstraction for appropriate representation. This enables engineers to solve problems on a conceptual level rather than thinking in terms of a programming language or paradigm. If abstraction was not involved, software engineers would have to delve into irrelevant details very early in the development cycle (Kiczales 1991).

Even though *Abstraction* existed as a general concept, a specific interpretation of it was brought forth by Fielding to explain the disconnect in the social dynamics of an organization in interpreting the same instance of abstraction. With ongoing distributed development, a single library implementation was unsuitable as different “forks” of the same software existed on the network. To solve this problem, *network based APIs* were introduced. It formalized abstraction with well defined semantics, placing no restriction on application code aside from network requests. At the same time reducing dependency to just protocol design instead of the actual implementation of the design (Fielding 2000).

Network based APIs brought with it the architectural pattern of Representational State Transfer (REST). It brought with it the notion of a client-server architecture where the responsibility of maintaining state was solely on the client. This *stateless architecture* was preferred as it made the system scalable and interoperable. Each request in REST architecture was independent of other request and was unaware of the overall component topology. Designing the system using this architecture ensured that server interactions could be scalable as well as interoperable with other services (Fielding 2000). By relying on

low level and well established protocols, such as HTTP (Hypertext Transfer Protocol), REST allowed the internet to achieve enormous scale by allowing *hyper-media* transfers at low latency and reduced server load by eliminating the need for the server to maintain persistent sessions for each of its clients (Jakl 2005). The standardization of server protocols coupled with breakthroughs in software architecture patterns such as APIs were essential towards the development of *Web 2.0* (O'reilly 2007).

Aside from REST architecture, data serialization protocols were also standardized. Two dominant formats Extensible Markup Language (XML) and JavaScript Object Notation (JSON) emerged. The API responses are sent in these *human-readable* formats which allows for interoperability between two systems. Modern web applications use these formats to pass data within two modular pieces of the same application (Yates et al. 2014). These serialized formats allow for easy parsing as well include scope for *contextualizing* data. This ease in manipulation adds flexibility for interpreting and visualizing data in new and unique ways. This was exemplified when Google launched their Maps service and allowed for interesting application with data being sent in AJAX (Asynchronous JavaScript and XML) format (O'reilly 2007).

Following its formal introduction in 2000, APIs and REST were first brought to commercial use by Salesforce followed by another internet giant of that time, eBay. APIs have since transformed the web and mobile landscape. In 2002, Amazon launched Amazon.com Web Services API, the precursor to the cloud computing behemoth, AWS, which has transformed IT architecture (Lane 2012).

Software Development

The challenges of Software Development have long been recognized and studied in IS literature. The idea that software development would always be a challenging endeavour was captured early in Brooks' (1987) classical paper, *'No Silver Bullet – Essence and Accidents of Software Engineering'.* To capture and address the enormous complexity of software and the issues regarding its development, several attempts at codifying 'methodologies' have been made (Iivari & Maansaari 1998; Avison & Fitzgerald 2003).

Although there has been a myriad of development methodologies, a good starting point to analyse the evolution of Software Development would be the Waterfall model. The key insight of this model is the splitting of development process into discrete steps or phases. The output of each phase is required for the completing of the next phase(Royce 1970; Boehm 1988). Although the Waterfall model had drawbacks and became unsuitable for the later era of Software Development, it remains an important methodology credited with formalizing Software Engineering.

Breaking away from the formal structure of Software Development, at the turn of this millennium, there was a push towards adopting *nimble* and *flexible* methodologies. These were termed as Agile and they were first conceptualized in the Agile Manifesto (Fowler & Highsmith 2001). There were many variants to the Agile methodology such as XP, Scrum, RAD, Crystal, etc. The common theme, however, was iterative development and a reduction

in analysis and documentation as compared to the older variants of Software development methodologies.

There have been various attempts to understand the notion of *agility* in the context of Agile software development in IS literature. The continual readiness to adopt change rapidly, proactively or reactively, and to learn from it has been termed as agility (Conboy 2009). Other key characteristics of Agile development have been regarded as incremental, cooperative, straightforward and adaptive (Abrahamsson et al. 2002). Even though agility was initially introduced in the context of software development, researchers have applied this concept at a firm and organization level as well (Lee et al. 2006; Sambamurthy et al. 2003). Similarly, this essay, in part, takes the idea of coordination which has been examined at the organization level and applies it to the realm of Software Development.

Recently, as Software Engineering has evolved, certain tenants of Agile methodology do not work effectively (Turk et al. 2014). One such drawback is the ineffectiveness of Agile methodology to incorporate distributed teams. A central theme in methodologies based on Agile approaches is high degree of interaction and communication between developers. However, an increasing trend of using teams that are geographically distributed has brought to question the requirement of extensive interaction among developers.

The composition of software development teams has changed from the traditional agile setting. With the segregation of tools and technologies for backend and frontend development, it is no longer feasible to group these teams into a single Agile team (Dinakar 2009). Frontend development is focused around User Experience and deploys client side technologies such as HTML, CSS and various JavaScript frameworks. On the other hand, backend development revolves around data and deploys server side technologies such as databases and server frameworks (Spring, Flask, etc.) (Van Waardenburg & Van Vliet 2013). Thus Agile development occurs within frontend and backend teams instead of a single clubbed team.

Coordination

Coordination has been extensively examined in ICT literature using a variety of theoretical lenses. The following section presents a brief summary of key theoretical contributions that are relevant to this essay.

Malone & Crowston (1994) define coordination as the process of "managing interdependencies between activities". They present a rational perspective towards the idea of coordination as a cost. Performing a microeconomic analysis, they argue the use of ICT to replace more expensive human mediated coordination. They argue that coordination heavily depends on *intention* and *incentives* among actors that are coordinating. Since these incentives and intentions are easier to describe and contain in technological artefacts, coordinating using ICT is cheaper and easier.

In Malone & Crowston (1990), they present coordination theory as a set of principles about coordinating *activities* and how *actors* can work together harmoniously. This theory emphasizes the role actors performing shared actions and highlights the role of

dependence. It then defines the role of breaking up actions into tasks to be performed by various actors and allocating resources. Another important aspect of coordination theory is information sharing to achieve common goals.

Aside from providing an economic rationale to explain coordination, they also bring a managerial view to coordination theory. Organizational aspects of coordination such as managing shared resources and maintaining producer-consumer relationships form key parts of coordination. An area of heavy focus is managing various kinds of dependencies among actors. After identifying the various areas where coordination is required Malone & Crowston (1994) suggest various strategies to handle temporal and task based dependencies. They also highlight the use of various tools that facilitate coordination.

Malone & Crowston's (1990) rationalist view of coordination has been criticized for not addressing various social embedded phenomenon. Their assumption of perfect incentive alignment among various actors as well as discarding other human factors have been listed as key shortcomings. They also assume the availability of enough information about the environment to predict and characterize interdependencies (Faraj & Xiao 2006). Another major drawback of their work is not addressing unplanned coordination and contingencies that arise in the process of coordination (Heath & Staudenmayer 2000).

An alternate view of coordination for fast response organizations was suggested by Faraj & Xiao (2006). They defined coordination as "integration of organizational work under conditions of task interdependence and uncertainty". While their approach shares the organizational paradigm for explaining coordination, they differ in their analysis of task dependency. Instead of assuming a pre-existing organization design to handle tasks, it looks at the ability of actors to coordinate task as they happen. This is highlighted in an alternate definition of coordination presented by them:

> "temporally unfolding and contextualized process of input regulation and interaction articulation to realize a collective performance"

Faraj & Xiao (2006) heavily focused on the idea of skill coordination. They argue that in fast response organizations, such as medical centers, expertise is distributed among various members of the team. Thus to coordinate, information sharing and shared cognition is necessary. They found evidence that expertise coordination enhanced performance by ensuring availability of crucial information throughout the team. They suggested various mechanisms that organizations deploy to achieve skill coordination. These included, protocol reliance, fluid teams and knowledge externalization.

They also recognize the existence of *Dialogic Coordination Practices.* Dialogic practices include continuous interactions, joint sense-making, varied event interpretation, and the existence of boundary objects. By recognizing the situated nature of coordination, they address the possibility of unplanned coordination and contingencies they may arise. On the downside, this view of coordination cannot be generalized to structures and organization that are more organized.

Okhuysen & Bechky (2009) provide an *integrative* perspective towards the idea of coordination. They define coordination as the integration of independent tasks which is central to the purpose of the organization. In the spirit of such a definition, they differ from

Faraj & Xiao's idea of managing uncertainty and instead focus on emergent practices that assist coordination. The define five mechanisms which encapsulate the idea of emergent coordination – roles, objects and representations, plans and rules, proximity and routine.

Assigning critical value to the idea of plans and rules under a coordination paradigm, Okhuysen & Bechky align to Malone & Crowston's view of a formal structure in an organization. Designating tasks, allocation of resources and developing agreement are essential parts of plans and rules present in an organization. Objects and representation aid in coordination by providing information. They provide boundary information to allow teams and organization to make decisions and smoothen the coordination process. Roles represent the relationship between people and the social structure present in the organization. Okhuysen & Bechky's idea of roles play well to Faraj & Xiao's idea of knowledge sharing and aiding team effort to as means to improve coordination. Extending the concept of knowledge sharing and transforming them into stores of knowledge is achieved by assigning routines. This helps in providing a template for completing task while aiding in building towards a common perspective to work towards.

Okhuysen & Bechky (2009) argue for three integrative condition essential towards the idea of coordination. These conditions – *accountability, predictability and common understanding* – address certain demands imposed on individuals due to the uncertainty created using coordination. Interestingly, each of these conditions can be achieved independently and via various means. Their framework thus allows for creating integrating conditions where specialized coordination is a requirement.

Ascribing to the practice view in coordination, Constantinides & Barrett (2012) promote the view that interdependency is poorly understood in coordination. The emphasize the need to break from explaining coordination as just physical acts and instead focus on the multiplicity of actions taking place in the narrative. Their view of coordination employs the use of 'narrative networks' to explain the interconnectedness in actions of various entities coordinating towards a common goal. Narrative networks consisting of both human and non human actors are constructed to explain *actions* between *actors* and *actants.*

Their approach provides a more situated understanding of coordination by maintaining different narratives of activities provided by the actors. By contrasting and comparing these narratives, a better temporal understanding of the coordination process is obtained. They argue that a narrative approach gives researchers scope to analyse the tension between fluidity and fixity or between improvised practices and shared models. They also allude towards a sociomaterial approach towards coordination. Sociomateriality states that human and non-human actors are inherently indistinguishable and the differences between actors arise from their interpreted attributes and capabilities (Orlikowski & Scott 2008). This temporal view, combined with Pickering's mangle of practice has been extended further by Venters et al. (2014). They argue that using a sociomaterial lens, the dynamism in digital coordination can be captured by viewing the various factors affecting the actors and material agencies involved.

Recent attempts in IS literature to understand coordination have moved away from modelling coordination as an organizational process towards a more situated view. A situated view allows for analysis of actions and actors present in a system which in turn

gives us a deeper understanding of coordination. Thus, it provides a framework for emergent forms of coordination and gives an alternative perspective for knowledge sharing inside the organization. However, by giving priority to practice over mental models, we can only capture a modified conception of knowledge which is mediated by both human and material agencies of the system (Gherardi 2008).

Theoretical Framework

This section provides a theoretical framework that would be later used for further analysis. Primarily, it involves summarizing the key facets of Coordination Theory presented by Malone & Crowston (1990). It also includes Okhuysen & Bechky's (2009) analysis of the integrative conditions required for coordination.

Coordination Theory sets out to unify ideas about coordination developed under various disciplines such as Computer Science, Management Science, Psychology, Economics and Organization Theory. It is defined as the body of principles about how the activities of different actors can be coordinated (Malone 1988). This section would summarize key elements of coordination theory under the light of modern software engineering.

The preliminary step in coordination is identifying *goals* that need to be achieved. These goals serve as ends for the coordination process. An important characteristic of goals in Coordination Theory is that they need not be shared among various actors. Having partly conflicting goals, in practice, is quite common. The presence of conflicting goals brings into question the evaluation criteria for coordination. Coordination Theory suggests that when analysing situations, the behaviour of actors should be analysed in terms of how well it achieves *certain* goals (which may or may not be held by actors). Thus, evaluating coordination is subjective to the observer and is dependent on their interpretation of the *goals* involved.

Activities are obtained by the process of *goal decomposition.* These activities are not independent in nature and it is this interdependence which requires coordination for completion. Thus both identifying tasks and the interdependence involved are critical in analysing coordination. The concept of *Abstraction*, as discussed before, applies when decomposing goals into activities. The level of detail when specifying activities is instrumental when analysing coordination. Each actor could be assigned several small tasks and this would help in obtaining a lower level picture of how these '*smaller*' tasks are carried out vis-à-vis assigning larger tasks to actors would bring out coordination at a higher level. For instance, analysing software development tasks at a technical level – database deployment, UI/UX design, etc. would bring out coordination among software developers. Alternatively, decomposing software engineering tasks at an organization level – requirement elicitation, deployment, etc. would bring out coordination among software developers and users.

Actors perform *activities* to get closer to the *goal.* To analyse coordination, it is important to identify the actors involved. This allows for the mapping of activities to various actors and bring accountability to the system (Okhuysen & Bechky 2009). However, identifying the various actors involved is often challenging. The degree of skill differentiation serves as an

effective indicator of whether different actors should be viewed separately or as a group within a system. In their analysis of medical centers, Faraj & Xiao (2006), group doctors who have varied skillsets as a single actor in the system. For the purposes of this essay, software engineers are broadly divided into two groups of actors – frontend engineers and backend engineers. Where frontend engineers focus on the *look and feel* of the software, backend engineers focus on the logic and data that drive the software.

The final component in coordination is understanding and managing interdependence among actors performing activities. These interdependences can be analysed in terms of *common objects* that are in someway involved with actions requiring coordination. For instance, in a scenario where software developers share computational resources, the interdependence amongst them can be analysed in terms of their common understanding of the same resources.

Malone & Crowston (1990) have identified three *preliminary* kinds of interdependence – Prerequisite, Shared Resource, Simultaneity. Prerequisite interdependence exists when the output of one activity is required as an input for another. For instance, in the Waterfall methodology of software engineering, requirement specification is a prerequisite for product design (Boehm 1988). Shared Resource interdependence is caused when multiple activities require the same resource. For instance, in modern software development, different modules of an application might use the same sub-module for a particular task. Simultaneity interdependence arises in situations where the temporal order of activities is a requirement. This is widely seen in distributed server architecture where there is a requirement for synchronizing relevant data amongst the various servers and they need to be kept updated in a simultaneous fashion (Dikaiakos et al. 2009).

According to Malone's Coordination Theory, there are four processes underlying coordination. It is useful to characterize these processes in a hierarchical fashion where each of them is dependent on the process above it. It is analogous to abstracted layers where each level presents different amounts of information and consequently lead to differently detailed analyses.

The first process in Coordination involves identifying the various components involved. These include actors, activities, goals and mapping activities to different actors. This is followed by Group-decision making – for organizations to effectively coordinate, some form of agreement between actors is required. This could be achieved through mechanisms such as consensus building, voting or authority. The third underlying process involved is communication. It is key in reaching agreement among groups trying to coordinate. However, to effectively communicate, establishing a common language and delineating the medium of communication is necessary for coordination. After establishing a common language, the effectiveness of communication depends on the actors' ability to perceive common and shared objects.

Accountability addresses the question of responsibility for specific elements of task. Okhuysen & Bechky (2009) argue that assigning responsibilities clarify which independent parties would be held liable for tasks. Breaking away from the use of accountability as means of exercising formal authority, they broaden the scope and the means of accountability by viewing it as integrating condition.

Even though Accountability addresses the question of *who* performs which tasks, Predictability is required to answer the question of *when* they are performed. Predictability gives a sense of which subtasks would be performed and in what order and is an essential integrative condition. They argue that being able to predict when certain tasks would be performed by their counterparts allows coordinating individuals to plan their own work in a coordinated fashion.

The final integrative condition pointed out by Okhuysen & Bechky (2009) is the notion of *common understanding*. When participants in interdependent activities share a common perception of goals and objectives along side knowledge of work that is to be performed. Objects and roles as discussed above aid in advancing common understanding among participants performing independent tasks. Having a broader overview of objectives allows organizations to align themselves to a common goal (Pinto et al. 1993).

Research Context and Study Design

Research Context

WeWork Moorgate is a co-working space located in Central London, United Kingdom. The largest of its kind in whole of Europe, it rents out office space to small and medium sized startups in London. Having the capacity to host over 3,000 entrepreneurs, WeWork Moorgate provides a staggering picture of the London startup culture. A majority of startups working here have a high technology focus and are building a business around a digital artefact.

Startups interviewed for this essay were of small to medium variety in terms of operation and size. They had a small dedicated team of engineers working on a software platform which was critical to their business. Most of these teams were organized around backend and frontend functions. Some even had Data Scientists integrated with their engineering team, this observation fell in line with the growing popularity of Analytics in software development at large (Kim et al. 2016).

One qualifying requirement for startups that were interviewed was that they developed and maintained an API. The requirement that these APIs also be open i.e. exposed for use by anyone outside the organization was not imposed. This notion breaks from the conventional use of APIs as *'gateways'* for third party developers as engineering teams are increasingly using APIs internally for effective scaling.

The startups that were interviewed were developing digital products in some capacity by themselves. Since these startups working on new products, they were working with very little technical debt. Without the cost of supporting legacy systems, these startups had the option to build upon newer cutting edge technologies. This allowed them to experiment with newer organization settings and adopt novel software engineering approaches.

All interviewees were either the CEOs of their organization or heading the engineering function in the capacity of CTOs. Given that these startups had business critical components

tied to the digital artefact they were working on, interviewees had a huge stake and interest in the product. This allowed for a germane discussion about how the engineering effort was organized internally. They had an overview of the entire product as well as the broader significance of the product and the API in an organizational capacity.

Study Design and Research Methodology

Since this research aimed to identify the internal Software development processes of London based startups and their usage of APIs as coordination tools, it took an interpretive approach. This approach has enjoyed growing popularity in the Information Systems field due to the multidisciplinary nature and its requirement to study different concepts. Thus an interpretive approach is suitable for this research to help link up Software Engineering practices with the literature on Coordination. Another advantage of using an interpretive approach is that it provides an overarching socio-technical view of the phenomenon while keeping in context the various social influences that affect the situation itself (Kaplan et al. 1994).

Another key decision when defining research methodology is to justify a qualitative or a quantitative approach. A quantitative methodology largely involves statistical analysis and collection of numerical data (Cornford & Smithson 2006). A quantitative approach usually works with positivist epistemology as it is ideal for objectively testing hypotheses. However, an interpretive approach calls for a more qualitative analysis (Cornford & Smithson 2006). Hence, due to the complexity of the phenomenon being studied, an interpretive approach along side qualitative data collection and analysis was undertaken.

To analyse the usage of APIs as a coordination device among these startups, a case study approach was undertaken and a series of semi structured interviews were conducted with key figures from these startups. A case study approach is argued to be effective in organizational studies where questions of why and how were asked to the interviewees (Yin 1994).

Yin (1994) argues that a case study approach is a useful strategy when investigating "contemporary phenomenon with its real life context, especially when the boundaries between phenomenon and context are not clearly evident". This is especially true when studying software engineering practices at startups due to the relatively unstructured and chaotic nature of their operation.

To enable generalizability for a case study, Yin argues for the use of a conceptual framework since theories are more general than cases. Malone's Coordination Theory was chosen as both for underpinning this research and analysing the collected data as well as further operationalizing the data. The reason for choosing Malone's Coordination Theory over modern views on Coordination such as Faraj & Xiao's (2006) situated and temporal unfolding approach was the relatively organized nature of Software Engineering. As there were fewer high-response situations and lesser demand for ad-hoc coordination among teams, Malone's Coordination Theory was apt in explaining coordination dynamics at these startups. Moreover, Malone's framework explains how organizations coordinate and the tools for analysing coordination in more detail rather than what Coordination is. This was suitable to study coordination among modern software development teams in startups.

Semi-structured interviews conducted to collect data across eight different startups working in different industries provided a rich insight in Software development practices and the use of APIs in particular. Galletta (2013) points out that the key benefit of conducting semi-structured interviews is its attention to lived experiences while keeping in close context theoretically driven variables of interest.

Another advantage of conducting semi-structured interviews is that it allows following a rough outline of themes with the possibility to explore topics of interest that may arise during the course of the interview (Cornford & Smithson 2006). Due to the conversational nature of the interview, there is scope to ask for clarification or further explain something if there exists ambiguity. As a result, semi-structured interviews are effective tools to understand nuanced and specific scenarios (Cornford & Smithson 2006).

Interviews were conducted with key people of various startups located in WeWork Moorgate working with APIs. The interviewees both had a technical and a managerial insight into the product they were developing. In total, nine interviews, each lasting between 30 minutes and 65 minutes were conducted. They involved startups that were working in varied fields such as:

- Social Media Analytics
- Internet Telephony
- Event Management
- Financial Portfolio Management
- E-Commerce Platforms
- Advertising Platforms
- Enterprise SAAS Solutions

Interviewing startups in these diverse fields revealed common insights and challenges in Software Engineering across each of these industries. Interview recordings were thematically analysed and were grouped into categories (Galletta 2013). Following this, these categories were analysed using Malone's coordination theory and certain aspects of Okhuysen & Bechky (2009) integrative take on coordination.

There are certain, well documented drawbacks of interpretive case study analysis using semi structured interviews. They include questions of generalizability along side alongside the validity and potency of data collected. These limitations were kept in mind during study design and would be discussed in the final chapter.

Findings and Analysis

The following section presents key findings and analysis obtained from the interviews. Key themes that were recognized and deemed relevant to the essay help in categorizing the findings. The first observation was an uptick in the adoption of APIs in the industry. The next finding analyses the role of API as a shared language. The theme of independence afforded by the use of APIs was another major finding during the interviews. This is followed by the observed limitations regarding the use of APIs as coordination mechanisms. The section ends with an overview using the processes underlying coordination as mentioned in Coordination Theory and the integrative conditions required for coordination as claimed by Okhuysen & Bechky (2009).

API Adoption

Interviewees were asked about what they thought of APIs and their significance in the Software Development at large. Respondents overwhelmingly expressed excitement around the *'API Economy'* that the industry largely has been moving towards (Anuff 2016). One respondent exclaimed:

"If you're not building APIs now, what are you doing?"

Interviewee 3

Their enthusiasm stemmed from the possibilities that APIs opened up in the scheme of software development. One respondent, building an internet based telephony solution claimed that their product would not have been possible without the availability of social APIs:

"In the past you couldn't have made a call to LinkedIn, made a call to Facebook, done a Google Search and compiled results inside your application for each (user) request"

Interviewee 7

Another interviewee expressed the ease of integrating with cloud computing platforms:

"... thousands of Amazon (Web Services) requests are fired today to load one web page"

Interviewee 2

Aside from the possibility of integrating their product to third party services, another prospect of considerable excitement was the open nature of APIs. By open nature they meant that both the APIs and the terms of use of these APIs were available on the internet for anyone to see. This added layer of transparency between two organizations transacting in large amounts of information is unprecedented. As one interviewee expressed:

"As a new startup, trying to move people from the traditional world to the cloud world, we are trying to adopt systems that are open"

Interviewee 5

Even though there was unanimous agreement upon a push towards API in consumer applications, some respondents expressed that Enterprises have been reluctant towards adopting an API style architecture. Their reluctance can be explained by the fact that an API architecture introduces fair amount of uncertainty and risk in application that Enterprises are averse to. One respondent explained:

"We want enterprises to open up to APIs but most are not satisfied with the security they offer"

Interviewee 2

These findings largely align with the literature around APIs. APIs have become industry standards in a short amount of time due to their ability to separate implementation from interface (Fowler 2002). As discussed before, they are instantiation of software abstraction and successfully the exposed endpoints and the internal implementation. By exposing APIs, companies can empower thousands of developers in a way where developers can expect certain functionality from an API without worrying about its implementation or taking on the liability to manage it. Consequently, APIs can develop without any cost to its users (de Souza & Redmiles 2009).

API as a shared language

According to Malone's coordination theory, an essential requirement for coordination is a common language between actors. This low level requirement is often a challenge in coordination. It is worth noting that the word 'language' has been used in coordination literature in a very broad sense. For Malone & Crowston (1994), developing a system of communication, including formal and informal methods as well as cooperative work tools counted as forming a 'language'. This has been observed in Faraj & Xiao's (2006) study of a medical trauma center as well. They point to a system of shared protocols used between the staff as a communication tool. With this idea as its motivation, the effectiveness of APIs as a shared language between developers was investigated.

One respondent when quizzed about how APIs are used as a communicating device within their organization said:

"The handshaking mechanism between the frontend developers and backend developers is the API schema"

Interviewee 2

The word 'schema' was used consistently across interviews by respondents. By API schema, they meant a formal definition of the API. Its formal definition included several components. The primary component was the API endpoint, a HTTP address wherein the API was located, the required and optional parameters that would be accepted as well as the expected response for a request that needs to be then handled by the client. Not only was the

schema well structured it invariably followed the REST standard and returned responses in JSON format. This standardization ensured that switching costs between different APIs were low.

Given how critical API documentation is, API designers take it very seriously. A majority of startups that were interviewed had adopted a new technology for documenting their APIs called *'Swagger'.* Swagger is capable of auto-generating documentation for REST based APIs directly from the source code. As one respondent puts it,

"We keep all our APIs synced with Swagger to ensure our documentation is always current"

Interviewee 9

Any changes in the API source code is automatically reflected in its documentation. The importance of documentation is, indeed, well documented in IS literature (Parnas & Clements 1986). Thus by automating the documentation process, there is minimal human interference in establishing what is anyway a tightly controlled and well defined formal language. Thus a language with very little room for ambiguity and high degree of transparency is established between developers.

Another essential feature of the common language established between actors, is their ability to interpret the language. The ability of API users to rely solely on the documentation provided was investigated. Most respondents thought of it as a non issue:

"These developers generally know what they are doing"

-Interviewee 1

Given that the final users of APIs are software developers themselves, important assumptions about their knowledge base can be made. Primary of these assumptions being that software developers are technically adept in using the API. This assumption also draws strength from the fact that API protocols are largely standardized in form of REST architecture.

This situation can be analysed from Faraj & Xiao's (2006) knowledge focussed perspective of coordination. They argue that when actors in a group coordinate, they contribute their individual knowledge repository to a high group level cognition. To illustrate this, they point to doctors in a medical trauma center pooling in their knowledge to treat patients where individually their knowledge would be insufficient. Their analysis breaks down in the kind of coordination occurring at these startups. No assumptions regarding software developers pooling in their knowledge while coordinating can be made. The only requirement for developers to coordinate is their shared understanding of the API schema as discussed above.

A recurring theme in both the interviews conducted and their subsequent analysis pointed to APIs reducing the amount of communication required for coordination. There was interesting conflict observed between reduction of communication and the Agile software development methodology they followed. One respondent claimed:

"We are truly Agile; we don't just pretend to be Agile"

-Interviewee 3

As pointed out in the previous section, a key part of Agile development is effective communication within the team. For instance, the Scrum method includes activities that require high amount of coordination such as sprint planning meetings and daily scrum meetings (Schwaber & Beedle 2002). Given that most of these startups were following an Agile approach, the independency and seclusion offered by APIs broke away from the ideal. When probed further, it was found that these startups were following a granular approach to Agile. Both the frontend teams and the backend teams were following them within each teams rather than following it collectively.

Even though APIs being used internally reduced the need for communication significantly, it was not an effective tool for group decision making. According to Coordination Theory, group decision making is critical for successful coordination. Agile approaches were largely used to build consensus in groups and make the inner-workings of the backend and front end teams more transparent. However, once consensus was reached, due to cleanly divided nature of the teams, arriving at task and subtask dependency was trivial. As one respondent puts it,

"In case of changing requirements, we look at the affected portions and developers required to update their code get on to it"

-Interviewee 4

API Independence

"The beautiful things about micro-services and APIs is that it naturally leads itself to independence"

-Interviewee 3

The theme of independence afforded by APIs was consistently brought up by respondents. When analysed further, there were a few key reasons why API brought this sense of independence in the context of Software Engineering.

The primary reason to use APIs internally was to promote *Parallel Development*. Essentially, the response returned by an API does not have to be a "real" response. By a process referred to as 'stubbing', backend developers could define the endpoint, generate a schema and return placeholder data. One respondent explained this process in detail (emphasis added):

"The first job is setting up a stub API while UI/UX guy is working on some sketches, API developers and UI developers come in and agree to the end-points we need, we create the end-points even if they are not *live"*

-Interviewee 2

When the respondent was asked what he meant by the API not being live, he explained that the data returned by the API is fake for the the time being and when the API implementation is complete, it would start returning actual data. Most startups were using a stub implementation of APIs to allow backend and frontend developers to work in parallel,

however, a social media analytics startup expressed stubbing as a critical part of their operation due to a third layer of data scientists being involved in the development process. While data-scientists decide *what* data to show, backend developers work on database connections and other relevant infrastructure while the frontend developers work on *how* to show the data.

The idea of stub-implementations is not a new one. This has been referred to as the contractual nature of APIs in IS literature (de Souza & Redmiles 2009; De Souza et al. 2004). APIs are viewed as contracts; two developers can work independently because the frontend engineer knows what is to be delivered by the backend engineer – it is specified in the API schema. Once everyone agrees on the contract, the stub implementation of API takes place. Eventually, at later stages of the the development cycle, the stub code is swapped with the real code and the API becomes 'live'.

This idea of independence fits directly in Malone's core idea of coordination – managing interdependence. Malone argues that if there is no interdependence, there is nothing to coordinate. Thus, it is imperative to establish the nature of interdependence as well as its existence in the paradigm of API independence. There is, in fact, dependency existing when developers choose to coordinate using APIs. This is observed when there is a need for developers to coordinate and agree on the API schema. This is a direct function of what Malone & Crowston (1994) refer to as Group decision making – actors (developers) communicating in some form (API schema) about the goals to be achieved. It should be noted that consensus building between developers is only regarding the interface specification. This is much shallower than achieving consensus regarding implementation details.

Having established the existence of interdependence, further analysis using Coordination Theory would reveal the kind of interdependence. To best understand a shift in the kind of interdependence, it is helpful to first understand the kind of interdependence existing in Software Engineering approaches that do not use APIs. In such approaches, developers first develop the backend infrastructure such as readying the databases, servers, etc. Only after completing basic work on such infrastructure is it possible to move to the Interface design. Such a workflow shows a *Prerequisite* interdependence. The work on the interface of the application is dependent on the output of the basic infrastructure.

In a software engineering paradigm where APIs are used, a shift from *Prerequisite* interdependence to *Shared Resource* interdependence is observed. As both the backend and frontend developers only share the API endpoints to coordinate, they can work in parallel. Again, the resource being shared here, the API endpoint, is an abstraction of an actual resource. When Malone & Crowston talk about *resource sharing* they implicitly assume it to be an artefact that all actors would deem similar. For instance, a machine being shared by two groups in a factory is the same machine for both these groups. However, in the case of APIs, even though developers share the API endpoint, it is merely an address to the *actual* API. The API itself is different for both the groups.

The idea that APIs are an abstraction of a *shared resource* is highlighted again in the following argument of interdependency management. When talking about managing shared resource interdependency, Malone & Crowston point to several disciplines handling the

same problem. In Economics, resource allocation is studied from a market perspective (Smith & Nicholson 1887). Organizational Theory has studied power dynamics associated with the act of resource allocation (Salancik & Pfeffer 1978). According to Malone & Crowston, the biggest problem in managing shared resource interdependencies is allocating these resources in the first place. However, this is mostly a non-issue when it comes to sharing API endpoints. Nether power dynamics nor market forces are required to explain how an API endpoint should be shared. The only explanation for the same is that API endpoints are not actual resource in *themselves* but an abstraction of a resource that is being shared. The biggest problem in managing *Shared Resource* interdependency in the case of APIs is *not* the allocation of resources but their maintenance. The problems of maintaining APIs would be discussed in a later section in detail.

Aside from parallel development, APIs bring another design shift in software engineering – Modularity. Parnas & Clements (1986) suggested the idea of abstraction and modular implementation where the implementation details of software should be hidden. APIs are an instantiation of the same principle (de Souza & Redmiles 2009). One respondent claimed:

> "Building our software around APIs allows us to move from a monolithic architecture to a micro-services (architecture)"
>
> -Interviewee 3

This shift towards a micro-services architecture has been an industry wide trend (Dragoni et al. 2016). The startups interviewed were overwhelmingly building software using this architectural paradigm, one respondent explained:

> "If 90% of your iPhone app is built in house and 10% of it requires a module that you don't have the expertise to build, having used micro-services comes in handy"
>
> -Interviewee 8

Another advantage of structuring engineering efforts around the API/micro-services paradigm is that it allows for effective distributed development (Turk et al. 2014). By having modular components, organized around APIs, coupled with the possibility of parallel development involving low requirement for communication, makes it possible for geographically distributed teams to effectively manage complexity.

Limitations of API

Managing API complexity

In investigating API as a coordination mechanism in software development, certain limitations of APIs were also observed. APIs become increasingly complex over time and managing them requires significant resources. Although, in theory API cuts down on communication, poorly implemented and managed APIs require significant support for usage. There is a high degree of lock in related to APIs as complex software is built around it. Aside from this, developers run into practical limits of integration. Moreover, the rigid

boundaries defined by API are sometimes disadvantageous and hinder development. These limitations would be subsequently presented and analysed in the following section.

The intent behind abstracting functionality into modules and wrapping them into APIs is to manage software complexity. Ideally, the API documentation should be the only communication necessary between API producers and API consumers. However, managing granularity is a challenge. On asking respondents how they managed API complexity they said:

"... you have to be clever with DevOps (Developer Operations)"

-Interviewee 1

"the honest answer is ... poorly"

-Interviewee 9

It was observed that once APIs grew to a sizeable number, it became really difficult to keep track of them and ensure they wouldn't cause conflict with the other APIs. One respondent, the CTO of his startup, who had joined the organization six months back, elaborated:

"When I joined ... we had 3 different APIs to just send E-mail, I had to clean up the (API) mess before adding new functionality"

-Interviewee 8

Another major problem with managing APIs was that as code evolved, it changed the API and consequently the API schema had to be updated. However, if the API users did not upgrade code on their end as well, the API would break the application. This problem was aggravated in organizations that were offering their API to developers outside their organization. This made evolving the API a cumbersome process. On the flip side, when startups were using APIs not built inside their organization, they were constantly fearing that the third party would change their API and break their application. One respondent expressed this sentiment:

"[Social Media Website] have great APIs but they keep changing them every five minutes. Their company is about moving fast and it reflects on their API"

-Interviewee 6

Two solutions to these problems were presented. The first one, *Versioning,* involved creating multiple versions of the same API and tying the API version to both the request and the response. Depending on the version, the server returned appropriate response. However, this put additional stress on the backend and added complexity to API management. The second solution was called *Overloading.* It involved changing the schema to only add newer response elements but never take the old ones away. Doing this ensured that the applications relying on the old response elements would keep working without breaking. This however made the the API schema confusing by returning multiple copies of the same data. As one respondent summarized:

"You can only add (response elements), never take any away"

-Interviewee 9

Both the solutions were partial workarounds to the above problems. This situation analysed using Coordination Theory shows another interdependency among the actors – *Simultaneity* interdependence. This interdependence emerges when the temporality of actions is a consideration. According to Malone & Crowston, the biggest challenge in managing Simultaneity interdependence is *synchronisation* of activities. This can indeed be observed in the API limitations mentioned above. Developers struggle to keep code bases that are distributed in synchronisation. Thus new coordination challenges emerge under the API paradigm.

API lock-in

An issue of API lock-in was a source of major concern among respondents. Since the final product was deeply integrated with a particular API, there were high switching costs if the API had to be replaced. This was especially true for APIs providing a service that was highly specialized. Due to the nature of their service, there were no standards to adhere to in terms of API schema. This made the lock-in problem even deeper. One respondent that provided a telephony API to third parties expressed:

"Once they (third parties) use our APIs, we have got them as customers for life"

-Interviewee 7

On the flip side, one respondent belonging to a startup heavily dependent on an API for their business expressed his concerns:

"If I am paying for their API, [Social Media Website] have an obligation to keep it same forever"

-Interviewee 6

The lock-in problem is not trivial and respondents claimed it to be a major reason why enterprises were sceptical to hop on the API economy. This problem has been referred to as API instability (De Souza et al. 2004). In practice, after parties agree on the API schema, it is often difficult to stick to the same specification once the coding starts. As new requirements emerge, the API schema needs to be updated, and this causes problems with developers who have built their code around the old schema.

Changing requirements hindering Software Development is not a new phenomenon. Interesting parallels regarding the same can be drawn from the disadvantages of the Waterfall model; the assumption that requirements can be analysed and documented perfectly turned out to be problematic. Malone's Coordination Theory is, perhaps, inadequate to explain the temporally unfolding situations where contingencies arise in such scenarios. However, these situations arise only after API schema turns complicated and is mismanaged.

Rigid Boundaries

APIs are very good at drawing rigid boundaries between development teams. Arguments presented in the previous section show that teams that use APIs to coordinate among themselves have reduced amounts of coordination. The rigidity of the boundaries creates a coherent organization structure around APIs that might be detrimental in certain situations. As one respondent explained:

> "... we have seen problems arise when the same API is used across two modules, we need to ensure that an API that works for both the modules does not cause conflicts in any one of them"
>
> -Interviewee 5

There was also a lack of awareness of the overall product direction. Developer resources could not be shared outside the rigid boundaries. Developers kept working inside siloes without much awareness or input regarding the finalized design and product. An even bigger problem was that without a deep knowledge of how code was implemented in other teams, their skills could not be put to much use outside their teams. This was observed in part across all startups that were interviewed. However, possibly due to the small size of their teams, this was not seen as a very big issue inside the organization. When probed about the same a respondent said:

> "Our developers are good at what they do, we do not need our backend guys to support our front developers"
>
> -Interviewee 1

This problem was aggravated when startups used third party APIs. Since the only communication medium was the API schema, an ill defined schema caused major issues. One interviewee expressed his concern:

> "If the API documentation is good, it is not a problem. For the APIs without proper documentation, it is not worth it chasing their engineers"
>
> -Interviewee 6

Another major condition with this problem is that since API documentation is supposed to be self sufficient, organizations providing APIs to third parties are hesitant to provide support. The metaphor of an API as a contract breaks down in these cases.

The problem of rigid boundaries was addressed in Faraj & Xiao's (2006) analysis of dialogic coordination practices as well as cross boundary intervention. According to them, in situations involving high stakes, having fluid boundaries helps in achieving high degrees of coordination as cross boundary expertise and knowledge sharing becomes critical. Thus, even though rigid boundaries are advantageous in making coordination simpler and effective, they are disadvantageous in the event of a coordination failure.

Integrating conditions for Coordination

Okhuysen & Bechky (2009) in their analysis of coordination give an alternative perspective to Coordination Theory. Citing lack of generalizability and a dearth of explanation regarding "how" coordination happens, they present a framework explaining the integrating

conditions required for coordination. They argue that all the various mechanisms that appear frequently in Coordination literature(Malone & Crowston 1994; Goodman & Leyden 1991; Faraj & Sproull 2000) can be grouped into three conditions for coordination: Accountability, Predictability and Common Understanding. In this section, to provide an alternative perspective, Coordination using APIs would be analysed using Okhuysen & Bechky's integrative conditon framework.
Accountability refers to the responsibility taken by actors to perform certain tasks. By making responsibilities visible, actors are held liable for their actions as well as making others liable for theirs. This can be achieved through both formal and informal devices. In the case of APIs, Accountability is built into the schema, well defined variables like speed, scalability, security and robustness hold APIs and by extension, its developers, accountable. Accountability ensures that an overall picture of everyone's contribution is visible to the group.

Predictability refers to the expectation that interdependent actors can ascertain when certain tasks would take place and how they are work towards achieving the common goal. This ensures that actors can plan and perform their work accordingly, which according to Okhuysen & Bechky, is essential for coordination. APIs ensure Predictabilty in two ways. Firstly, by being rigidly defined, they leave little room for ambiguity about the function between interdependent actors. Aside from cutting ambiguity, due to the stubbing of APIs, they act as finished artefacts from the very beginning of the its lifecycle. This ensures developers to plan around it assuming its availaiblity.

The third integrating conditon, Common Understanding, refers to a shared understanding of how each actors' works fits into the picture. Okhuysen & Bechky argue that Common Understanding is high whena actors share their work and knowledge. Aside from this, having a shared knowledge base aligns actors towards their goals. APIs fall short on this integrating conditon. By cutting down on commincation needs, they restrict sharing of knowledge among backend and frontend developers. This also results in little alignment towards a common goal. Even though formally APIs do not help in building a Common Understanding, startups often employed Software Development techniques like Agile that contribute highly to the same. By ensuring a healthy discussion regarding objectives, developers aligned to the common goal. However, APIs proved an hindrance as they only shared an understanding about the goals on a superficial level and not at an implementation level.

Processes underlying Coordinating using APIs

As discussed in the previous section, there are four processes underlying coordination. Each of these processes are best understood in a hierarchical fashion. This section aims to break down the process of coordination using APIs into these four processes based on the findings and analysis done previously.

The first process underlying Coordination involves identifying goals, activities, actors and assigning actions which are formed by the process of goal decomposition to actors. In the case of Software Engineering, this step involves establishing requirements regarding the Software artefact under construction. Furthermore, it involves identifying developers responsible for fulfilling the requirement. The next step involves assigning tasks and

subtasks formed by decomposing the requirements to appropriate developers. In the context of APIs, a higher level understanding of the API requirements is sketched out.

The second process underlying Coordination is Group Decision Making. This step involves building agreement among actors and coming to a shared understanding of the tasks involved. In the context of APIs, this is done by coming to consensus regarding the functionality of APIs. Both backend and front developers agree on the functionality and the modular nature of APIs. The granular nature of APIs allows for code reuse. The disadvantage of granularity is the added complexity it brings. Developers need to agree on the appropriate trade-offs and come to an agreement.

The third process underlying coordination is establishing a common language between actors and deciding the medium of communication. This common language as discussed before, is the API schema. The medium of communication between frontend and backend developers is the API documentation. As discussed before, when developers agree to use APIs, they cut down on the need to communicate heavily and the technical documentation of APIs serves as the shared language.

The final process underlying coordination if the understanding the perception of common objects among actors. This involves the objects that are shared and how the interdependencies among actors should be managed. This shared resource, is the API itself. Initially, it involves sharing a stub API, which is then transformed into a fully functioning API. However, as noted before, APIs are an abstraction of the shared object. The interdependency to be managed here involves maintaining the API so as to satisfy both the front end and backend developers.

Conclusion and Discussion

Conclusion

This essay has analysed the effect Application Programming Interfaces have on Coordination in the context of Software Development. This analysis was based on the literature and interviews conducted with a few startups based in London. Despite an intuitive sense of APIs acting as coordination mechanisms, the findings reveal a more nuanced picture of how APIs are being to better coordinate among software developers.

The analysis which was primarily done using Coordination Theory (Malone & Crowston 1994), reveal a strong reduction in overall interdependencies and point towards a more structured form of coordination. A shift in software development from prerequisite interdependence to a shared resource interdependence was observed. Practically, APIs were shown to facilitate parallel development and support modern software engineering constraints such as continuous integration and distributed development.

During the analysis, key inferences about the nature of APIs were also drawn. The abstract nature of APIs along side its modular property brings about different effects on coordination. Since APIs are abstract resources, they do not fit the mould of common shared resource. This brings about a key shift in how interdependencies around APIs should be

managed. The modular nature of APIs is a double edged sword, it is an effective tool for organizing development and reusing code while at the same time they bring in a lot of complexity than needs to be managed.

On the whole, the use of APIs in the development process brings in a lot of structure and aids in coordination. They lead to a higher degree of autonomy by eliminating the need for humans to participate in the coordination activity. However, as in Software Engineering there are '*No Silver Bullets'*, the use of APIs come with certain limitations. They lead to a lock-in in both a technical sense as well as organizational sense. Moreover, for all the structure they bring about, they turn out to be disadvantageous in situations of high uncertainty.

Owing to explosion in use of APIs, further research should be carried on their adoption inside organizations of different industries and sizes. Secondly, APIs should be studied as an IS artefact by themselves. They converge IS literature regarding modularity, generativity, abstraction, software development, open innovation and spill over economics to name a few. As a central topic to this essay, research regarding coordination and the role of API can be studied in different organizational contexts at different scales.

Limitations of the study

Software Engineering is a complex discipline involving various stakeholders. The focus of this essay is to study coordination between – frontend developers and backend developers – two very specific groups involved in the Software Engineering process. It does not cover coordination inside these groups or other critical people involved in the software engineering process. Thus, the findings of this essay are only applicable to understand the coordination between these two groups.

Another glaring limitation of this study is that involved observing small and medium sized startups located in London. The software development occurring at these startups is not reflective of the traits and characteristics of software development occurring in different contexts. The startups interviewed for this essay were had relatively small software development teams and were less bureaucratic when compared to larger organizations. Thus, they presumably had very different problems regarding coordination. This means that the findings of the study cannot be generalized to different kinds of organization.

The use of semi-structured interviews for collecting data bring about certain limitations. The interviewees, by virtue of being key stakeholders, are often biased towards certain aspects. This brings into question the validity and honesty of their answers. For this particular essay, respondents were often hesitant to give a complete picture of the coordination and communication problems they faced. The qualitative information provided is often riddled with context, thus drawing generalized conclusions often misrepresents what interviewees were trying to express. Aside from this, there is an underlying risk of the interviewer, by being aware of the research data and the overall literature, influencing the answer of the interviewee subconsciously.