A Multi-Agent System Architecture for Carpooling Solutions

Brendan Devenney

Submitted in partial fulfilment of
the requirements of Edinburgh Napier University
for the Degree of
BEng (Hons) Software Engineering

School of Computing

November 2015

Authorship Declaration

I, Brendan Devenney, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines

Signed:

Date:

Matriculation no: 40053530

Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below *one* of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

The global social, economic and environmental state lends itself to high adoption rates of participatory culture schemes such as carpooling. There has been significant government discussion around carpooling in the United Kingdom in recent years but no significant action taken to support or encourage uptake. Corporate solutions such as SAP and Comovee pose a high financial barrier to entry in some of the most troubling economic times. Meanwhile, the lack of research concerning modern carpooling system architecture creates a barrier to in-house development and deployment of such systems.

The aims of this project are: to investigate the factors for and against adoption of carpooling systems and schemes; to architect a system which takes into consideration these factors; and to investigate the benefits and drawbacks of agent technology in such a system. Evaluation and validation is conducted via a proof of concept implementation of the architecture which is based on the scenario of Edinburgh Napier University's staff taxi-sharing when travelling between campuses.

The major discovery of the evaluation was a clear bottleneck in the Java Agent Development Framework (JADE) due to the directory facilitator offering significantly worse scaling than the rest of the framework in systems high on communication. This problem has previously been solved through search caching per container – a solution which is not applicable for a truly scalable carpooling system. The paper proceeds to provide a novel solution to this problem which builds upon the previous work.

In conclusion, agent technology is suitable to carpooling solutions due to the high distributability of the problem and the clear matching of the agent lifecycle to the lifecycle of the carpool negotiation process. Due to the unforeseen focus on the directory facilitator bottleneck, the paper concludes with a wealth of further work which would build upon the work presented in this paper.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my project supervisor Dr Neil Urquhart for his tireless efforts to support and improve this project, Prof Emma Hart for her prophetic warnings on scope issues which the project repeatedly faced, and the collective staff of Edinburgh Napier University for their teachings which culminated in this paper.

I would also like to thank my family and friends: my parents, Kevin and Valerie Devenney, for their continued support and for sparking my interest in technology when they purchased my first computer; my partner Laura Gunn for supporting me endlessly despite my year-long affair with the Java Agent Development Framework; and my close friends for understanding my repeated absence from social events.

1. Introduction

1.1. Motivation

The benefits of carpooling are multiple: social, economic and environmental. Yet existing corporate solutions such as Comovee (sysware, 2015) and TwoGo by SAP (2015) are expensive and inflexible, targeting large corporations which are willing to front the costs of bespoke, branded software – while inexpensive open solutions such as BlaBlaCar (2015) fail to provide closed groups within which carpools can be negotiated. Parallel to this issue for small to medium organisations, Artiach et al. (2010) found that the most significant factors in leading corporate sustainability performance are: organisation size, higher levels of growth and higher return on equity.

This combination of high cost and low corporate sustainability performance make it very unlikely that small to medium businesses will take the initiative in deploying a sustainable ridesharing scheme for their employees. In scenarios where cash flow is a factor against purchasing such software – or even in the scenario that employees wish to run their own carpooling system – there appears to be no acceptable solution.

1.2. Aims and Objectives

The primary aim of this project is to explore the feasibility of an abstract agent-based carpooling architecture which could be adopted and integrated into a corporation with minimal effort. Given the autonomous nature of intelligent agents, the project will seek to discover the benefits and drawbacks introduced: perhaps abstraction to the point of simply specifying parameters and fitness functions will be possible – equally, it is possible that the complexity of agents makes an abstract system extremely complex to develop.

From the aforementioned come the high level objectives:

- Understand the factors for and against the adoption of carpooling systems, ensuring the architecture includes and excludes these respectively;
- Review successful carpooling architectures – agent-based or otherwise – to identify their strengths and weaknesses;
- Discover the most suitable technologies for deployment, management and scalability;
- Architect an abstract carpooling system which empowers developers to deploy software which improves corporate sustainability;
- Develop this extensible architecture with hooks wherever business logic would be required to determine the system's behaviour;

- Implement a use-case of this architecture in the form of a taxi-sharing system between the three main campuses of Edinburgh Napier University;
- Validate and evaluate the architecture using this proof of concept implementation.

1.3. Chapter Outlines

Chapter 2: **Literature Review –** An overview of the problem domain; factors for and against adoption of carpooling systems; an analysis of existing solutions, their advantages and their drawbacks; and suitable technologies for consideration in the design and implementation phases.

Chapter 3: **Requirements Analysis** – Identification of the functional and non-functional requirements of the architecture and implementation, along with the identified technologies which will be utilised – all guided by the findings of Chapter 2.

Chapter 4: **Initial Design** – A preliminary system architecture design which describes the components, structure and relationships; a high-level design of the architecture's components; and a deeper design of the communication and sequences involved in carpool negotiation.

Chapter 5: **Implementation** – A full implementation of the architecture based on a realistic scenario which will be utilised for testing and validating the architecture. The aforementioned scenario will also be outlined in this chapter for clarity.

Chapter 6: **Evaluation and Refinement –** Stress testing and validation of the architecture to ensure that the implementation works and the architecture is feasible. Evaluation of the implementation's quality attributes against the identified non-functional requirements. Any modifications identified at this stage will be documented and re-evaluated in this chapter.

Chapter 7: **Final Architecture** – The final system architecture design based on lessons learned in implementation and improvements identified during evaluation. This section will include work from Chapter 4 which was not modified for the sake of presenting a complete architecture.

Chapter 8: **Conclusion / Further Work** – Conclusions on the application of agents in a carpooling system architecture; further work identified during the project which would improve the architecture or provide a stronger answer to the feasibility of agents in such an architecture; and a personal evaluation of the project as a whole.

2. Literature Review

2.1. Carpooling and Ride-sharing

As defined by Oxford University Press (2015a), a carpool is “an arrangement between people to make a regular journey in a single vehicle, typically with each person taking turns to drive the others.” From this concept came dynamic carpooling – or real-time ridesharing – which relies on recent technologies such as Global Positioning Systems (GPS) and networked mobile computers (Hess, 2011). Only officially acknowledged by the Oxford English Dictionary in February 2015 (TIME, 2015), ride-sharing is defined as “[participating] in an arrangement in which a passenger travels in a private vehicle driven by its owner, for free for a fee, especially as arranged by means of a website or app” (Oxford University Press, 2015b). This recent addition to the Oxford English Dictionary is a sign that the sharing culture has shifted: what was once a casual phrase is now a term of international stature, and a process which was once time consuming and difficult is now empowered by web applications and social networking.

Car sharing has become a focal point of discussion in the United Kingdom in recent years, with significant government investigation into the ways which law, city planning and government bodies could be restructured to encourage a sharing culture (Wosskow, 2014). While there has been a plethora of discussion, there has not been sufficient action taken to implement the ideas. As discussed by Baldassare et al. (1998), lack of carpool availability is the largest issue, with around one third of drivers expressing a strong likelihood to shift away from solo driving if the availability of carpools was expanded – with a further third showing a slight likelihood. More recent evidence of the effects of insufficient carpooling availability is given by Abrahamse and Keall (2012), in which it was discovered that around 61% of those who had registered for Wellington’s “Let’s Carpool” initiative had not started carpooling – with 49.1% of this group citing the reason as “still looking for better matches”. In contrast to static carpooling initiatives, informal dynamic carpooling has seen some success - the most notable being the casual carpooling (or 'slugging') culture that has established itself in Washington D.C, San Francisco and Houston (Federal Highway Administration, 2010). This approach to carpooling resides between hitchhiking and public transport: passengers queue at an informal meeting point and await the arrival of potential drivers. The slugging culture has seen success despite no money being exchanged: both parties benefit from casual carpooling, such as the driver avoiding tollbooth charges, passengers avoiding public transport fees or both parties saving time by gaining access to high-occupancy vehicle (HOV) lanes (Pisarski, 2006).

There is strong evidence that steadily rising fuel costs are related to significant increases in car sharing (Bento et al., 2013). Koppelman et al. (1993) found that the introduction of parking fees to carpool models greatly increased the likelihood of adoption, and predicted that emulation of these taxes “could come, conceivably, in the form of the new clean air regulations coming into play in the next few years, or perhaps as the result of major, long-term increases in energy prices.” When it is taken into account that Great Britain’s fuel prices consistently rank amongst the highest in the European Union having steadily increased year-on-year since 2009 (Bolton, 2014), it seems fair to assume that the likelihood of carpool adoption is extremely high in the current climate. This combination of government interest, predicted public acceptance and the fact that 39% of United Kingdom energy expenditure is due to transport (Department of Energy and Climate Change, 2014) places the environment strongly in favour – and need – of new carpooling solutions.

Perhaps then it is most important to design a system which accommodates the factors which have previously pushed users away from initiatives. Abrahamse and Keall (2012) found that the factors which are persuasive to those looking to start car sharing are “flexibility in carpool arrangements (31.8%), increased financial savings (21.9%) and more information about the matches (19.8%)”. However, the reliance on others’ schedules, cost arrangements and lack of car in case of emergencies (when not the driver) were a significant dislike. Other studies have shown that the aversion to having no car between journeys is not limited to emergencies, as those who require a vehicle for their job are less inclined to form carpools (Baldassare et al., 1998). This may suggest a desire for the ability to always be the designated driver, though it could equally be due to a desire to keep the work vehicle clean or simply be a result of company policies. On the other hand, there are members of carpooling initiatives who explicitly cannot be the designated driver – the obvious case being those who wish to carpool but cannot drive thus are switching from public transport as opposed to solo driving (Abrahamse and Keall, 2012), while the less obvious is users who use carpooling as a partial journey such as to and from a public transport station (Wosskow, 2014). It has also been observed that consumers put strong emphasis on independence and timely service in premium carpooling solutions – those in which a business provides the drivers and operates as a cross between a carpool and a taxi service – which implies a majority of consumers in fact prefer to have full control over carpool formation (Koppelman et al., 1993).

2.2. Agent Technology

As described by Jennings and Wooldridge (1998), an agent is an autonomous entity which encapsulates both state and behaviour, capable of autonomous action based on its observations of the system it resides in. The distinction between an agent and an intelligent agent is defined by the nature of this autonomy; an agent can be described as intelligent if it is:

- Proactive: “exhibit[s] opportunistic, goal-directed behaviour”;
- Responsive: able to perceive and react to changes in the environment; and
- Social: capable of appropriate interaction with other agents in order to complete, or aid in the completion of, tasks.

Agent technologies have been widely identified as beneficial in the solving of problems which can realistically be broken down into sub-problems, from spatial reasoning (Kray, 2001) to the retrieval of healthcare information from complex distributed databases (Hosam et al., 2010). The benefits in such problem domains are multiple: agents may provide solutions to problems which were previously “beyond the scope of automation” due to lack of solvability or unfeasible cost with existing technology; problems may be solved in a more efficient way (Jennings and Wooldridge, 1998); and they may provide system adaptability in a dynamically changing global environment which requires negotiation and/or cooperation between agents (Bond and Gasser, 1988).

Agents in the context of a carpooling system primarily represent the people who are looking to form a carpool. Such agents can become related through common needs: similar timing, origin and destination of intended trips (Cho et al., 2012). Some successful approaches such as that of Sghaier et al. (2010) have extended this to include Zone Agents which each represent a zone in the overall route optimisation problem, allowing the shortest path problem to be solved by distributing Dijkstra’s algorithm across many agents – which communicate weights when a path spans multiple zones – demonstrating the power of agent collaboration. It is clear that an agent-based approach is suitable for solving the carpooling problem due to its two relevant characteristics:

1. Responsive: Must be solved, ideally in negligible time, when an agent (user) communicates its desire to form a carpool in order to present the end user with results.
2. Distributable: Requires a calculation to be executed for each and every other agent (user) in order to determine, at least, the feasibility of matching – or at most the perceived value of said match.

From this concept of agents and relationships, the foundations of a social network can be seen. The agents are vertices and the relationships are edges. Many of the modern approaches to intelligent carpool system architectures have made good use of this concept to integrate real social networking into the algorithms and applications (Cho et al., 2012; Kothari, 2004). The agents in such systems are related not only by journey details but also by common characteristics such as gender, career or (income) class. Additional relations are then built through feedback on arranged carpools – negatives such as turning up late or positives such as the cost calculations being very fair – and ultimately fed back into the algorithm to constantly improve the strength of matches.

Many tools exist for the development of agent-based systems, however the market has narrowed as the domain has matured to favour pure execution environments such as Java Agent Development Framework (JADE) and Agent Development Kit (ADK) as this approach does not require the learning and usage of a new language (El Fallah Seghrouchni et al., 2009). A vast majority of academic work is biased towards JADE, perhaps due to its initial focus on academia rather than business (Bellifemine et al., 2001) which has influenced many of its useful features such as a GUI to control and monitor agents as well as very complete logging for debugging and analysis. Chmiel et al. (2004) were able to successfully run a demonstrator system with thousands of agents on antiquated hardware - 120 MHz processors and 48 megabytes of system memory - using JADE which partially proves its suitability to, and scalability within, a carpooling system in which the maximum number of agents is limited only by the population's adoption of such systems and the number of carpools these users wish to negotiate.

Cortese et al. (2002) investigated the scalability and performance of JADE's message transport system (MTS) and found that JADE provides a significant performance increase over remote method invocation (RMI) when agents requiring regular communication are grouped in the same container, while inter-container communication performance is roughly equal to that of simple RMI. The study concluded that correct agent distribution between containers, combined with the ability to migrate agents between containers before a period of high communication, makes JADE "a good candidate for developing heavy-load distributed applications." Further studies by Piccolo et al. (2006) found that the JADE framework scales linearly with regards to both agent migration and agent communication. Across-the-board linear scalability suggests that the JADE platform's throughput will be proportional to the available computing resources. However, some downsides have been detailed in more recent years: Mengistu et al. (2008) found that the realistic scalability of the JADE platform does not line up with the scalability of its core functions (message passing, agent migration) due to the inefficiency of the agent directory service - "because this service is used frequently by the other platform services, its inefficiency affects other services too.” Due to the carpooling problem’s high levels of communication, this may indeed be the biggest problem faced when scaling the system to high-end realistic levels.

2.3. Reputation and Image

The primary drawback of autonomous agents is their inherent lack of both overall system control and global perspective of their environment, thus agents require reliable methods of determining trust and delegating to other agents (Jennings and Wooldridge, 1998). Sabater and Sierra (2002) pioneered a revered reputation model which uses social network analysis to add a social dimension to reputation - adding this on top of the previously established concept of comparing contracts with their actual outcomes – in order to reach direct agent interaction reputation or 'image'. The social dimension built upon this is multi-faceted and combines a number of information sources:

- Witness Reputation: third-party feedback on the target agent, which has the downsides of potentially false or incomplete information when the human element is added. Due to this the agent must give importance to each piece of information “based on the social relations between the witness, the target agent and the source agent,” as well as the actual reputation of the witness.
- Neighbourhood Reputation: “the reputations of the individuals that are in the neighbourhood of the target agent and their relation with him” – in which neighbourhood is not a physical concept but rather a collection of agents which have strong cooperation levels with the target agent; and
- System Reputation: a default value based on the role of the agent with regards to its institution (for example a carpooler may view the driver of a public bus by default as generally late but unlikely to overcharge, or view taxi services as timely but certain to overcharge).

Sabater et al. (2006) later designed Repage: a reputation and image system based on the earlier work of Sabater and Sierra (2002) on ReGreT which allows stronger reputation calculations by “not only [calculating] the reputation values but also [analysing] the origins of the results and [reasoning] about them.” Additionally, Repage maintains the distinction between image and reputation when dealing with third parties – where ReGreT would simply merge it into one value – to allow a more accurate calculation as the agent is aware of the differences between what *it* believes and what *others* believe. A similar system was proposed by Khosravifar et al. (2012) which also built upon the multi-faceted reputation concept of

Sabater and Sierra (2002). This system contributed interesting factors to the reputation calculation: improved merging methods – that is, the methods which are used to merge incoming trust ratings – to consider the “proportional relevance [and freshness] of each reputation value” to avoid the sensitivity to noise which is inherent in the previous work’s approach; and “basic learning ability as part of [the agent’s] decision making process” to combat malicious or deceptive agents whose referrals cannot necessarily be relied upon.

These models are particularly applicable to multi-agent systems which put strong emphasis on the negotiation stage – as in the negotiation of carpool arrangements – to reach more desirable agreements as “the agent could [accept] a not so good offer knowing that at the end it would be good enough” and “arriving to an agreement sooner minimizes the risk of partners’ withdrawal” (Khosravifar et al., 2012). This is useful in the context of carpooling when considering the previously referenced case studies: a large percentage of those willing to carpool were unable to find suitable matches – a problem which can only be solved by introducing greater flexibility to the matching – or their matches had already established carpools and thus were no longer available.

2.4. Cloud Computing

The National Institute of Standards and Technology (NIST), a non-regulatory agency of the United States Department of Commerce, defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell and Grance, 2011). Further, NIST asserts that cloud computing is composed of five core characteristics: self-provisioned services on demand, broad access to capabilities over the world wide web, virtualised resources which are drawn from a pool rather than being physically provisioned, up- and down-scaling which can optionally be automated, and metered service which provides transparency for both providers and consumers. The benefits of cloud computing are numerous, particularly in the case of small businesses or new application deployment (Avram, 2014):

- Drastically reduced entry costs with zero upfront investment into hardware or physical storage space;
- Ease of deployment scaling, with no lead time on hardware and no transitory downtime or lowered throughput during scaling;

- Dynamic scaling with the above characteristics – often programmatically and autonomously; and
- Novel deployment concepts: 'burstable' computing resources which can operate in a cheaper low-power state yet retain the ability to maximise processing power if necessary, massive parallelism on many low-powered machines, and hybrid computing which offloads high loads to cloud services.

However, the perceived barriers to enterprise adoption of cloud technologies paint a picture of wariness and hesitation to migrate infrastructure to the public or private cloud before it has been tried and tested by others:

- Uncertainty about privacy and security;
- The true power of cloud computing relies on global high-speed connectivity, requiring “more sophisticated consumer products”;
- Cloud infrastructure providers have thoroughly tested the reliability of their own services but have yet to establish a real track record of reliability due to their young age;
- There is much uncertainty about the interoperability of cloud services with traditional services, and also between the different cloud providers;
- Hesitation in budgeting for cloud computing due to the potential hidden costs (helpdesk, maintenance, et cetera);
- Requirement for new skill sets in the information technology teams present within the organisation – requiring retraining or further hires; and
- Issues with the global nature of cloud computing – privacy, regulations, law and politics.

Amazon Web Services (AWS) (2015) has been named as a leader in Gartner's Cloud Infrastructure as a Service 'Magic Quadrant' for five consecutive years. In this magic quadrant, service providers are rated on both their completeness of vision (strong market understanding and/or a vision for changing market rules) and their ability to execute (how well the provider can demonstrate their understanding or implement their vision). The 'leaders' quadrant is reserved for those showing prowess in both of the aforementioned metrics, in which only Amazon Web Services and Microsoft Azure are present. AWS are rated "as having both the furthest completeness of vision and the highest ability to execute" with Microsoft trailing almost 25% with regards to the ability to execute.

2.5. Geospatial Database Technologies

Two main approaches exist with regards to geospatial databases: spatial database engine (SDE) middleware and spatially-enabled object-relational database management systems (ORDBMS). SDEs serve as APIs for writing and reading spatial data to supported databases, providing a simple solution but no granular control over the implementation. Spatially-enabled ORDBMS support abstract data definitions to organise spatial data which are stored and output as objects which the user can write types, functions and methods to support. Avoiding middleware "makes it easier to implement spatial operations and generally results in good performance," however "[the] architecture must be bound to a specific database platform" (Hall and Leahy, 2008). Spatial databases are particularly applicable to routing tasks given the massive amounts of data required such as connectivity, adjacency, restrictions on roads - speed limits, limited directions or turning - and order such as roundabout or motorway exits (Egenhofer, 1993). Ray et al. (2011) benchmarked the available geospatial technologies for PostgreSQL (PostGIS), MySQL and Informix, finding that MySQL and PostGIS were equal when averaged over the majority of tasks which are similar to the carpool routing problem - map search, map viewing, and line and area intersections. However, it was found that MySQL lacked direct support for many operations and the implementations required to simulate these operations resulted in many false-positive records. For this reason, PostgreSQL with PostGIS appears to be most suitable to carpooling solutions in which false positives have a direct negative impact on the end user. The tools provided by PostGIS can also be extended by open source libraries such as pgRouting (pgRouting Community, 2015) which provides geospatial routing functionality. This routing ability, the ability to store linestring geometry (an object representing a sequence of coordinates and their connecting lines), and the ability to find the shortest distance between a given point and the entire linestring, can be combined to create a carpool matching solution which incorporates complex matching such as detours to pick up suitable members in the middle of the journey.

2.6. Map Data Technologies

The two leading sources of map data are Google Maps and OpenStreetMap (OSM). While developing with Google Maps as a downstream service has its uses, particularly the assumed reliability and accuracy of commercial data, the restrictive licensing must be carried through to any system developed on top of Google's data (Google, 2015). OSM data is licensed under the Open Data Commons Open Database License (ODbL) which is share-alike with attribution, allowing any system built on top of OSM data to remain open (OpenStreetMap, 2015). On top of this openness, many high quality open source libraries have been developed: GraphHopper

(2015) is a highly efficient Java routing library which takes OSM data as its input by default and outputs useful data such as journey time for each mode of transport, and OpenLayers (2015) is a JavaScript library for map rendering with mobile support. Such libraries may be modified or included in systems as-is without decreasing the openness of the architecture.

2.7. Literature Conclusion

The current state of the economy shows the desperate need for a working carpooling solution. With studies showing around two thirds of solo drivers showing a likelihood to shift towards carpooling if availability was extended (Baldassare et al., 1998) , and one third expressing their desire for increased flexibility (Abrahamse and Keall, 2012), it is clear that any proposed solution must accommodate regular carpooling as well as dynamic ridesharing. The concept of prearranged carpools is not unreasonable to combine with dynamic carpooling – a carpool will often not fully utilise the capacity of the vehicle, therefore a dynamic carpooler may join the carpool for a single journey. This combination maximises availability for the dynamic carpoolers, all while ensuring the reliability and savings which was found to be of the utmost importance for static carpools (Abrahamse and Keall, 2011).

The carpooling problem can be optimally solved by distributing the majority of negotiation logic across a computer cluster. Due to the strong support for distributed computing and heterogeneous containers, agent technology is clearly suited to the problems involved in a carpooling solution – though its contribution towards minimising solve time makes its strengths most apparent in dynamic solutions (Sghaier et al., 2010). Although the ideal scenario (no agent thread ever being blocked by the processor governor) provides negligible negotiation time (note that negotiation time does not encompass solve time), the worst case scenario is also supported by the agent platform as has been demonstrated by studies implementing JADE on antiquated hardware (Chmiel et al, 2004). The linear scalability of the JADE platform has also been proven (Piccolo et al., 2006), making system load predictable given established system (or user) trends, peak times and other such metrics – suggesting that the cost efficiency of the system can be safely maximised without unforeseen instability. The reputation and image aspects can be fairly well defined in a system which represents humans as agents. The integration of existing social networks can allow reliable reputation in and of itself given the wealth of information which is available at present – for example mutual workplace history, education, hobbies and interests. JADE, being a Java-based framework, is also particularly appealing due to the heavy use of Java in libraries which operate on OpenStreetMap (2015) data.

Cloud technologies are beginning to mature and lend themselves by nature to applications which are horizontally scalable, and due to the linear scalability of the JADE platform it is realistically possible to implement auto-scaling rules to ensure demand is always met. The computing resources required are, theoretically, a linear function of the number of agents in the system (Piccolo et al., 2006). Although migrating old, business-critical systems to the cloud may be premature before the providers have an established track record, the benefits of cloud deployment for new non-critical applications in small to medium-sized businesses have proven to be huge factors towards adoption and acceptance (Avram, 2014). AWS (2015) is a clear leader in the cloud infrastructure as a service both in terms of market share and the execution of market vision. For this reason, AWS is the logical choice for the deployment of a novel agent-based carpooling system.

Map data from OpenStreetMap is ideal for the architecture of a carpooling system not only because of the open source license. The quality of the data is particularly strong in cities, to the extent that roads are classified such as residential, dual carriageway or highway - as well as one-way or private – and the data is often 100% complete (ITO World, 2015). As speed limits in the UK are determined by the physical features (e.g. dual carriageway, single track) of the road unless local councils deem the road to pose specific dangers, the GraphHopper algorithms to determine journey time (and subsequently the shortest path) will be exceptionally strong for UK data even in areas with limited completeness of actual speed limits.

Some of the more niche technologies of recent years could also be of use in the creation of a carpooling solution. The heavy reliance on databases suggests that geospatial extensions would be well suited to the task, with some well-established open source libraries available for core tasks such as geocoding, navigation and map display. Specifically, the PostGIS extension for PostgreSQL seems to be perfect for the creation of an accurate and extendible architecture without compromising on speed (Ray et al., 2011). OpenLayers (2015) can be used in combination with the output of GraphHopper (2015) to display routes and allow users to input data through maps. All of these combine to build a solid foundation for the routing aspect of a carpooling architecture which is robust, efficient and open.

It is worth noting at this point that the conclusions of this literature review with regards to specific technologies should not be viewed as restrictions of the architecture. The choice of database technology, map data source, routing implementations, deployment (cloud or otherwise) and perhaps even the agent framework used should not be coupled too closely to the architecture itself. These technologies have simply been identified by previous research and studies as the most applicable.

3. Requirements Analysis

Prior to designing the system architecture, it is necessary to identify a number of key requirements. The requirements should naturally emerge from the literature review with additional input from technology investigations. These should be split into domains then further split between functional and non-functional requirements. There are however a number of requirements which span all of the domains.

The high-level functional requirements are threefold:

1. The system must autonomously negotiate valid carpools on behalf of users.
2. Users must be able to influence the system by initiating carpool requests and choosing between the carpool offers.
3. The system must be persistent and redundant to ensure that agents will resume in the correct state and no information is lost.

The non-functional requirements are less clear. The quality of savings (in economic terms) is the major metric of system performance and should be comparable against the cost of each user travelling alone. Further non-functional requirements regard the user experience: the system must be responsive and the client application must be intuitive.

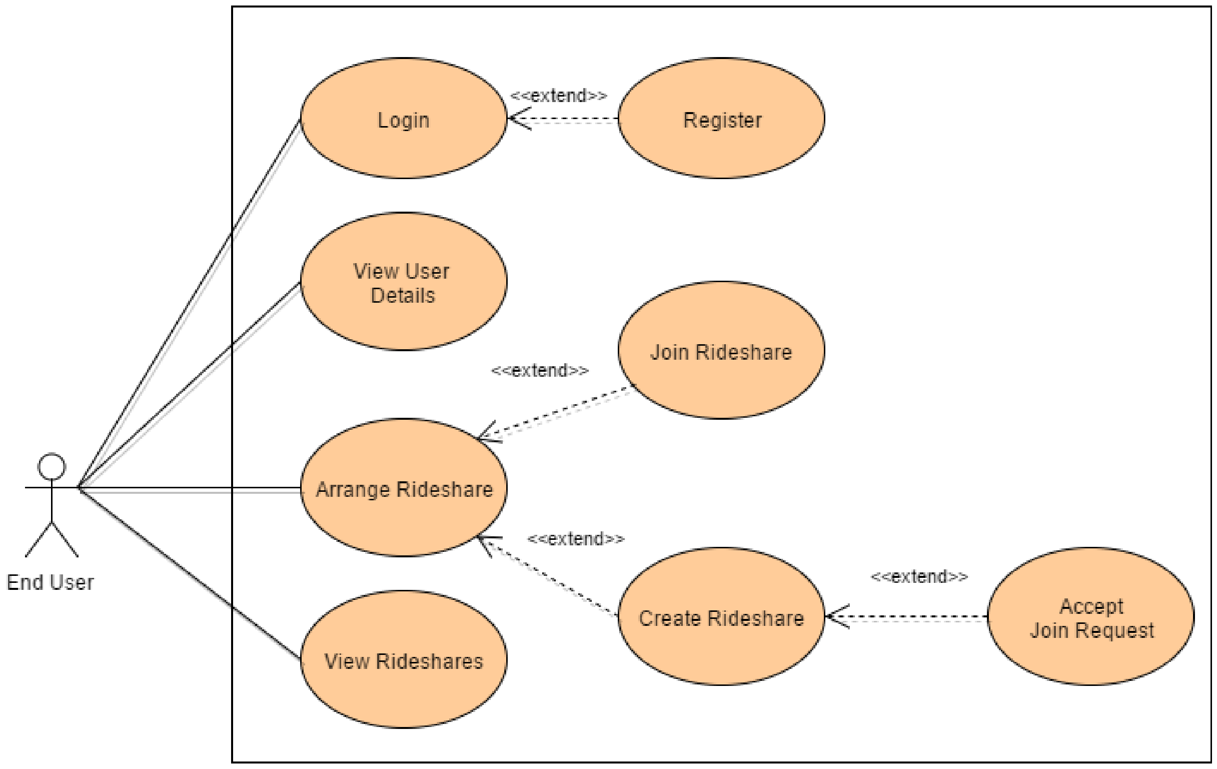


Figure 1: Carpooling Use Case

Additionally, there are some requirements enforced by the aims and objectives of the project which may not necessarily be optimal:

- All carpool negotiation must be conducted by an agent-based system; and
- The system must have additional monitoring, logging and evaluation tools to support this report which would not be acceptable in a true production system.

3.1. Technology

The high level functional and non-functional requirements dictate the technologies that must be employed to design and implement an architecture which is both fit for purpose and fit for use. The technology requirements can be logically split between the three functional requirements.

1. Autonomous carpool negotiation:
    a. To fulfil this requirement, the architecture requires an agent framework within which agents can communicate on behalf of their user. As detailed in Section 2.2, the best technology to achieve this was identified as the Java Agent Development Framework (JADE).
2. Persistence and redundancy:
    a. The required persistence of the system mandates a database which exists external to the agent system, using a database technology which supports replication and/or distribution to ensure that there is not a single point of failure. This was identified as PostgreSQL in Section 2.2.
    b. Agents will require database connectivity, conveniently provided through the Java Database Connectivity (JDBC) API. Java applications must programmatically load a JDBC driver which describes database interaction, generally provided by the developer of the database technology.
3. End user influence:
    a. In order for end users to influence their associated agents there must be a client application which can insert, update and delete data in the database.
    b. Developers will require a web application programming interface (API) for use in the client applications which provides controlled access to the data store.

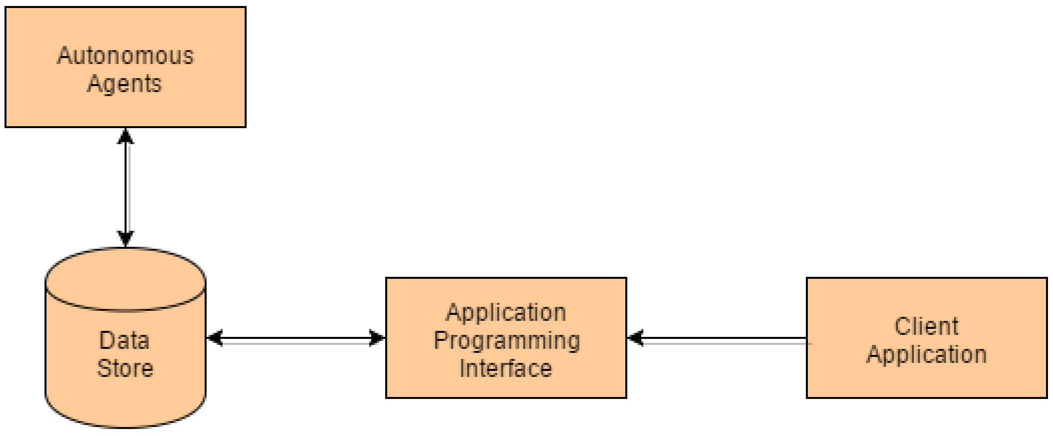


Figure 2: System Architecture Overview

Implicit to these technology requirements is the requirement to deploy these technologies on heterogeneous systems – in the cloud, on a local network or any combination of the two – in such a way that the relevant non-functional requirements are met.

3.2. Autonomous Agents

An agent system is composed of independent autonomous agents, each of which must meet the functional and non-functional requirements of the overall system. If every agent responds to a message in a time shorter than the defined timeout then the user experience is improved. Conversely, if a single agent is unresponsive then all queries including this agent will reach the timeout before continuing – a negative impact on the whole system.

1. Functional

- Carpool agents should respond with an offer to carpool seeking agents only when the date, origin and destination are the same *and* the time windows are compatible;
- Time windows should be updated to reflect the widest range which is suitable for all members to ensure that there are no conflicts between mergees; and
- Carpool agents should merge after successful negotiation to reduce total agents in the system from $\sum_{n}^{1}(members\ in\ carpool), where\ n = carpool\ count$ to $carpool\ count$, with the mergee deregistering with the DF Service for a clean termination.

2. Non-Functional

- Efficiency: Agent negotiation must be efficient to prevent system slowdown (as found by Mengistu et al. (2008), critical agents such as the DFService Agent can be overwhelmed) and maximise the number of agents per container before the container becomes unstable (the CPU cycles and Java heap space available to a single container is finite).
- Response Time: Carpool seeking must be responsive as the user expects to see a response in negligible time in the client application. This incorporates both carpool requests and offer acceptance – the two stages of negotiation which are initiated by and returned to the user.
- Recovery: All agents should resume in the expected state after a system restart. This requires a nonstandard entry point to the FSM which can transition to any stable state. States which are considered unstable are those which are awaiting responses from other agents in the system, as resuming in these states will result in a timeout due to the expected responses never arriving - which ultimately returns the agent back to one of the stable states.

- Robustness: Agents must *never* terminate as the result of a thrown exception as this is equivalent to a loss of information (specifically any uncommitted in-memory information, resulting in desync between the database and the system). Exceptions are tolerable and should be handled by falling back into the last known good state, informing the user if any of their actions were reversed in the process.
- Scalability: Support communication between distributed containers on heterogeneous systems, dynamically handling creation and population of these as necessary. This also applies to downscaling, thus agents must be able to migrate smoothly to containers on other systems before the system destroys them.

3.3. Data Stores

Data stores are necessary to allow users to influence the agent system while maintaining the definitive autonomy of the agents. In this scenario we can consider the data stores to be the observable environment of the agents. As such, the agent system is highly dependent on the data stores (but notably not vice versa) – without the data stores the agents will have nothing to observe or influence, but the data stores can tolerate agent system outages.

1. Functional

- The relevant details of all users and routes must be stored, along with the transient data required for communication between user and agent (such as carpool offers);
- Data must be updatable in place;
- An intermediary store must exist which links users and carpools, to fully support the concept of Route Agent merging; and
- Make the stored data visible over the system network (private subnet, virtual private cloud, or Internet) to authorised systems.

2. Non-Functional

- Availability: The data stores are a dependency of the autonomous agents and the application programming interface (API), therefore high availability system design is crucial.
- Disaster Recovery: Given that the only source of persistence in the architecture is the data stores, a loss of data can potentially reset the system to its initial state (with zero users and carpools). For this reason the architecture should consider the distribution and replication of these data stores along with the backup strategy.

- Performance: Negligible query time is necessary to prevent delays in agent negotiation and API calls. As the client application pulls all of its information from the API, non-negligible query times are extremely noticeable and detract from the user experience.
- Scalability: The data stores must be as scalable, if not more, than the agent system to ensure that it does not become a bottleneck as the rest of the architecture is scaled up.

3.4. Application Programming Interface (API)

The implementation of a web API is necessary to create a truly extensible architecture, ensuring that any changes to the backend only require a change to the API's wrapper code – not a change to the applications which consume the API. Additional benefits include tiered authentication, elimination of boilerplate code and simplified stack deployment on heterogeneous platforms due to the language-agnostic nature of web APIs. APIs can also be versioned to ensure that out-of-date client applications are not rendered unusable by business logic changes.

The API should be as decoupled from the database and clients as possible. This allow existing organisational databases to be hooked (such as pre-existing user databases) and arbitrary client applications to be implemented – from web clients to mobile applications, or even non-user monitoring/evaluation clients.

1. Functional

- Provide gated access to the backend for client applications, allowing users to:
  1. Authenticate
  2. View carpool details
  3. Initiate carpool searches
  4. Accept carpool offers
- Provide the service over the Internet to authorised users (via client applications).

2. Non-Functional

- Availability: If the API is unavailable, users are unable to influence the agent system. The backend will continue to negotiate carpools from the current state but users will not be able to view any uncached matches or accept any proposals.
- Flexibility: The API should be easily extensible to handle backend modifications and support new client application functionality.

- Integrity: Users (and their associated agents) should only have access to their own details, the details of carpools in which they are a member, and the details of other members of their carpools.
- Performance: Negligible response time is mandatory, otherwise client applications are unresponsive and the user experience is negatively impacted. This requirement is largely dependent on the data store performance and does not account for latency on mobile devices – the onus is on the user to ensure their connection is acceptable.
- Robustness: The API may be used by any software which can consume APIs. These applications may be developed by engineers who were not involved in the API specification, therefore it should handle incorrect usage correctly without polluting the data stores or compromising the data integrity.

3.5. Deployment and Scalability Solutions

A significant aspect of the architecture is the deployment and scalability of the production system. JADE provides a good starting point for distributed computing but provisioning systems for containers and deploying the other components of the system should be incorporated into the architecture.

1. Functional

- The backend must be deployable in the cloud or on a local server cluster; servers and their responsibilities should be customisable for true distribution.
- The system should be dynamically scalable without restart so as to not interrupt negotiations or disconnect users; and
- Deployment should be possible with a single user action: the central system should launch containers on remote systems and attach them to the main container.

2. Non-Functional

- Flexibility: It must be simple for system administrators to easily add or remove systems to the network;
- Performance: The system must come online in a reasonable amount of time, especially in such cases as system restart (for example when the Main Container must be relocated) and disaster recovery;
- Portability: Deployable on heterogeneous servers without issue. Servers may be in a local cluster, globally distributed or a combination of the two.
- Usability: Any aspects of deployment or scaling which require system administrator interaction must be intuitive and easy to use. It is likely that a perfect implementation

of the architecture would have a fully featured administration client – however scripts or an API would reasonably meet this criteria as the system administrator will be a technical user.

3.6. Client Application(s)

Just as the application programming interface provides application developers with an intuitive and standardised way of utilising the backend, the client applications provide users with a usable method of API interaction (abstracted to a level at which they require no knowledge of the API).

1. Functional

- Provide an interface to the carpooling system which utilises the API functions:
    1. Authenticate
    2. View carpool details
    3. Initiate carpool searches
    4. Accept carpool offers
- Cache details for offline viewing of carpool details, in case users need to see negotiated time windows or users when they don't have an active data connection.

2. Non-Functional

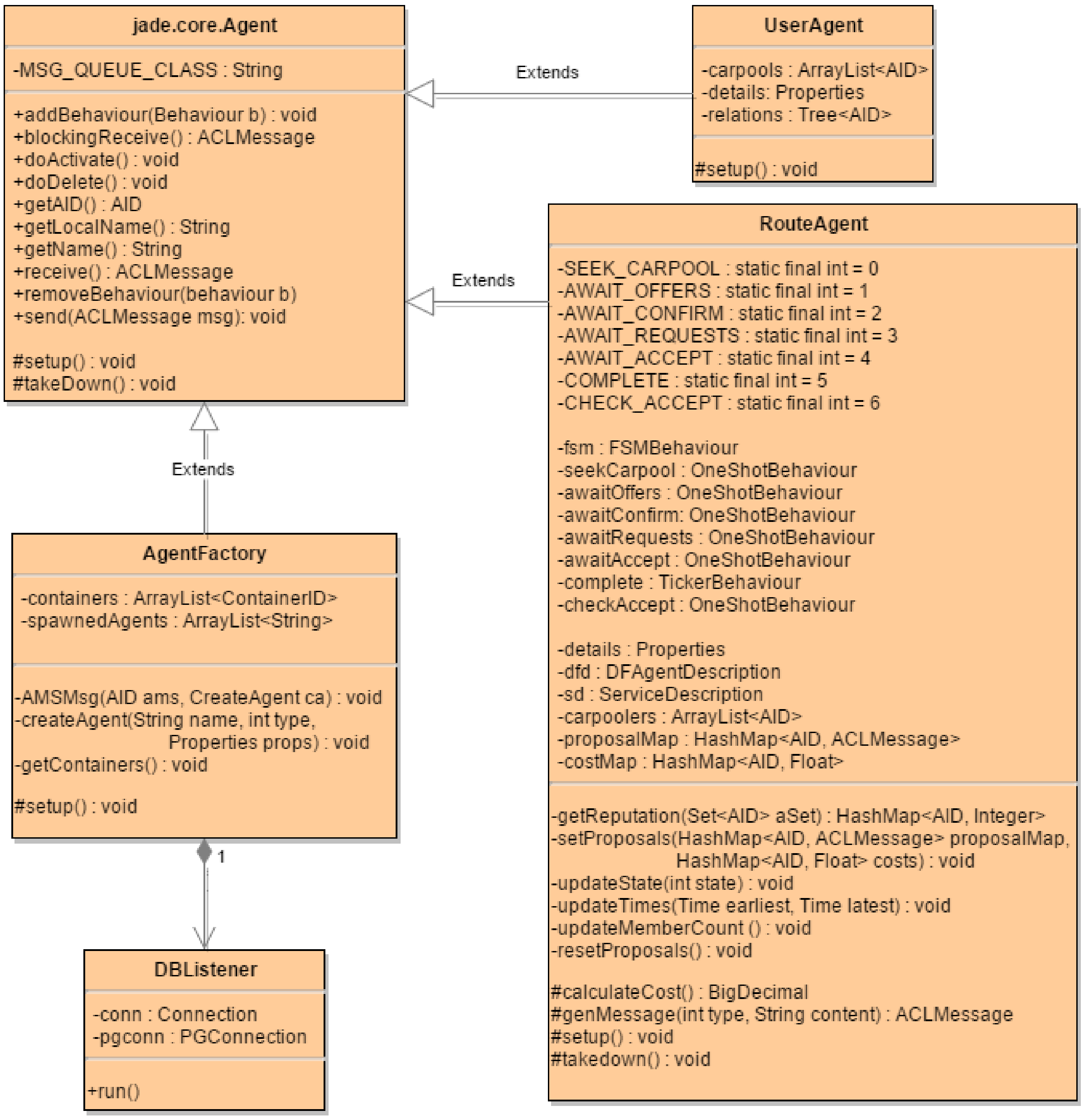
- Efficiency: The application must be usable on less powerful mobile devices, especially in the corporate scenario where inexpensive devices will likely be purchased to minimise costs.
- Flexibility: Easily modifiable on user devices as private carpooling systems are unlikely to have publically distributed applications (i.e. Google Play Store, Apple App Store).
- Integrity: User credentials should be stored securely. Traffic should be protected against man-in-the-middle attacks through use of cryptographic protocols.
- Robustness: Strong input validation is required to ensure that only correctly formed requests are sent to the API. Additionally, mobile devices may transition between cell towers and Wi-Fi or lose connection altogether – the client applications must be able to handle such unpredictable connections.
- Usability: Intuitive UI design with informative notifications is important as many users may be non-technical. A significant aspect of this is implementing as much input restriction as possible to guide the user in the right direction (e.g. using a date picker control instead of making the user manually type a correctly formatted date).

4. Initial Design

The carpooling system architecture is composed of three distinct layers: the backend with autonomous agents and data stores; the application programming interface (API) to provide client developers with backend access alongside the deployment and scalability tools; and the client(s) for end-user system use which can take any form provided that the platform is capable of API consumption. A number of the assumptions and decisions made here are due to the time and scope constraints of this project – further details can be found in Section 8.1 below.

4.1. Agent System

The agent system consists of three core agents and one optionally injectable agents. The core agents are concerned with the real-time carpool negotiation tasks such as agent creation, reputation, cost calculations and communication. The optionally injectable agent is concerned with providing debug statistics regarding the current system state, such as the total savings versus one-per-carpool. Realistically, there can be any number of injectable agents which can fulfil any desired task. The debug agent exists for academic reasons and should not exist in a production system – therefore the debug agent will not be included in the design diagrams.



jade.core.Agent
-MSG_QUEUE_CLASS : String
+addBehaviour(Behaviour b) : void
+blockingReceive() : ACLMessage
+doActivate() : void
+doDelete() : void
+getAID() : AID
+getLocalName() : String
+getName() : String
+receive() : ACLMessage
+removeBehaviour(behaviour b)
+send(ACLMessage msg): void
#setup() : void
#takeDown() : void
Extends
UserAgent
-carpools : ArrayList<AID>
-details: Properties
-relations : Tree<AID>
#setup() : void
Extends
RouteAgent
-SEEK_CARPOOL : static final int = 0
-AWAIT_OFFERS : static final int = 1
-AWAIT_CONFIRM : static final int = 2
-AWAIT_REQUESTS : static final int = 3
-AWAIT_ACCEPT : static final int = 4
-COMPLETE : static final int = 5
-CHECK_ACCEPT : static final int = 6
-fsm : FSMBehaviour
-seekCarpool : OneShotBehaviour
-awaitOffers : OneShotBehaviour
-awaitConfirm: OneShotBehaviour
-awaitRequests : OneShotBehaviour
-awaitAccept : OneShotBehaviour
-complete : TickerBehaviour
-checkAccept : OneShotBehaviour
-details : Properties
-dfd : DFAgentDescription
-sd : ServiceDescription
-carpoolers : ArrayList<AID>
-proposalMap : HashMap<AID, ACLMessage>
-costMap : HashMap<AID, Float>
-getReputation(Set<AID> aSet) : HashMap<AID, Integer>
-setProposals(HashMap<AID, ACLMessage> proposalMap, HashMap<AID, Float> costs) : void
-updateState(int state) : void
-updateTimes(Time earliest, Time latest) : void
-updateMemberCount () : void
-resetProposals() : void
#calculateCost() : BigDecimal
#genMessage(int type, String content) : ACLMessage
#setup() : void
#takedown() : void
Extends
AgentFactory
-containers : ArrayList<ContainerID>
-spawnedAgents : ArrayList<String>
-AMSMsg(AID ams, CreateAgent ca) : void
-createAgent(String name, int type, Properties props) : void
-getContainers() : void
#setup() : void
1
DBListener
-conn : Connection
-pgconn : PGConnection
+run()

Figure 3: Core Agent System Class Diagram

The three core agents are as follows:

1. The Factory Agent is responsible for the construction and delivery of agents.
    - Creation messages to the Agent Management System (AMS) agent are used to programmatically spawn agents.
    - On inception, data about all users and carpools is pulled from the database and used to respawn any agents which existed during a previous run (generally in the case of a system restart, though this method can theoretically handle a database from another architecture to migrate from a non-agent system to the architecture proposed in this paper).
    - There exists a PostgreSQL database listener which listens for insertion and deletion notifications on the user and carpool tables' channels, spawning and destroying agents to synchronise the environment.
2. The User Agent represents a single user.
    - User Agents hold a two-deep tree of related User Agents (direct relations which have formed successful carpools with their related user, and indirect relations which have formed successful carpools with the direct relations) to provide basic reputation within the system.
    - User Agents are also responsible for the removal of its user from all relevant carpools when the user is removed from the system.
3. The Route Agent represents a single carpool.
    - Route Agents communicate between themselves to autonomously retrieve the best potential carpools to present to the end user based on their requirements, ultimately merging together as a result of the end user's acceptance of an offer.
    - On its inception, it must populate a key-value list of all carpool properties passed to it by the Factory Agent as part of the agent creation message. These values may be as simple or complex as the implementation requires.
    - Agent behaviour is dictated by a finite state machine (represented in JADE as an FSMBehaviour which holds states, transitions and transition values). In general there are three states in which agents are not awaiting replies, therefore there are three states in which agents may be restored: SEEK_OFFERS, AWAIT_REQUESTS and COMPLETE. The initial state of the finite state machine is LOAD (not documented in the diagram for the sake of clarity) which may transition into any of the aforementioned based on the state which is held in the database entry. New carpool requests will set this to SEEK_OFFERS by default.

State	ID
SEEK_OFFERS	1
AWAIT_REQUESTS	2
AWAIT_ACCEPT	3
ACCEPT_OFFER	4
AWAIT_CONFIRM	5
COMPLETE	6

Table 1: Agent System States

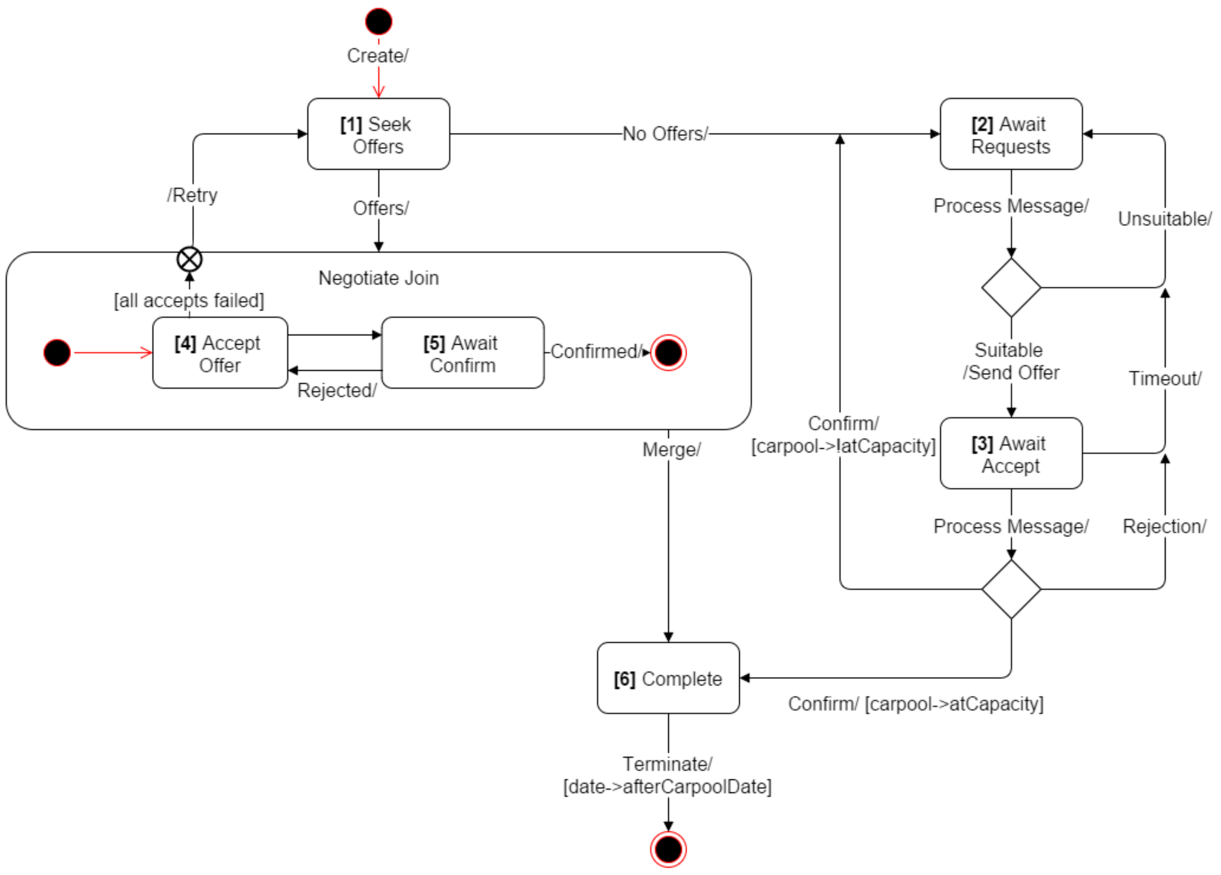


Figure 4: Route Agent Finite State Machine

The optionally injectable agent is:

1. The Evaluation Agent
   - Polls the database and provides metrics such as the cost of one user per carpool, actual cost of all carpools in the system, overall savings, or average savings per carpool.
   - Evaluations are executed as ticker behaviours, thus new metrics can easily be added and the polling frequency can be altered by modifying a single variable.

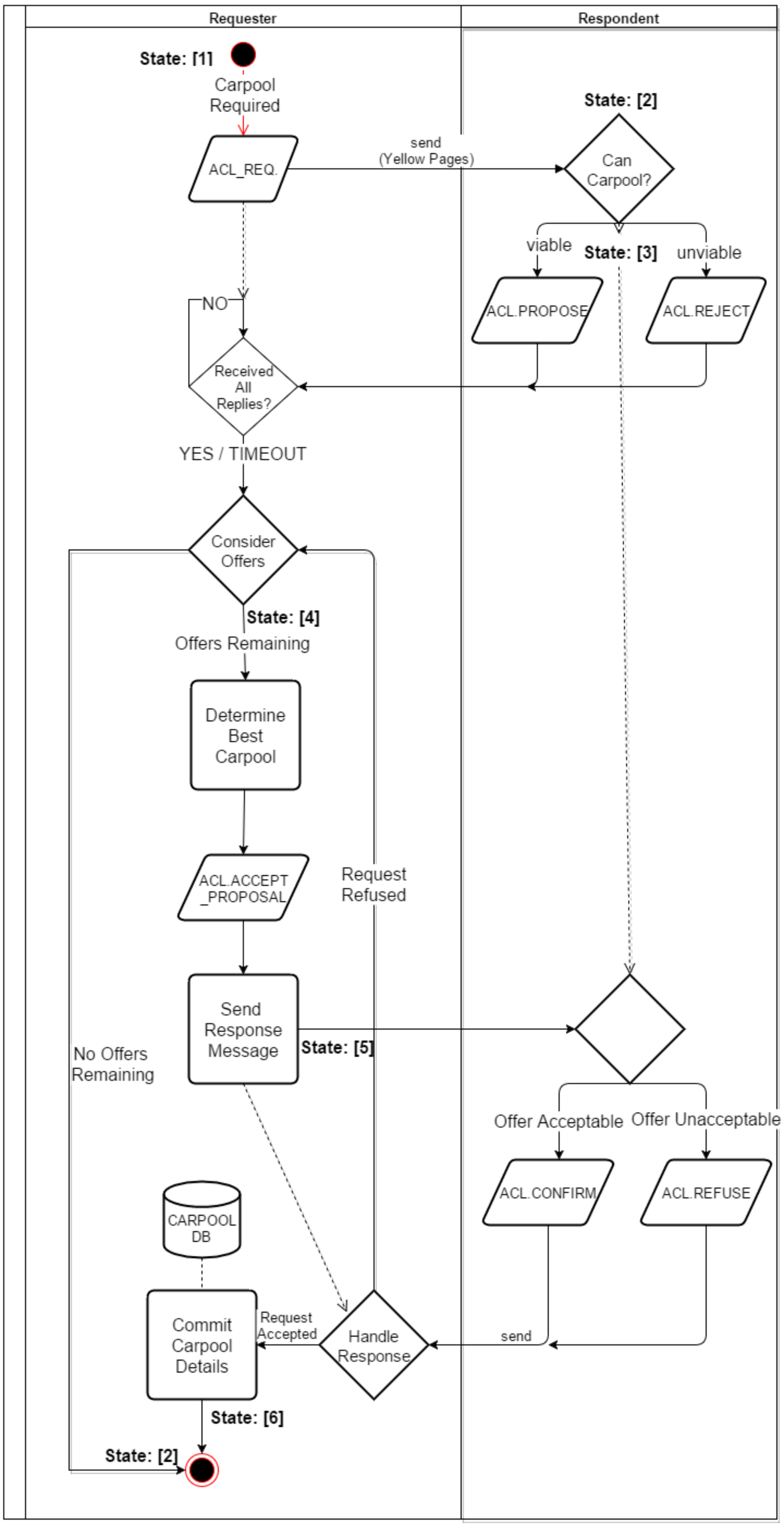


Figure 5: Carpool Negotiation Sequence Diagram

4.2. Database

The data store is fundamentally composed of two tables: users and carpools. One limitation of PostgreSQL is its lack of support for arrays of foreign keys. The solution to this issue is to use an intermediary table which links user IDs to carpool IDs, allowing for example to get all members of carpool 100 with the query:

```
SELECT uid FROM ucintermediary WHERE cid='100';
```

A similar table is required to store carpool proposals – however it is important to note that the intermediary and proposal tables cannot be combined into one, as this requires an extra field to denote whether the entry is a proposal or an arranged carpool.

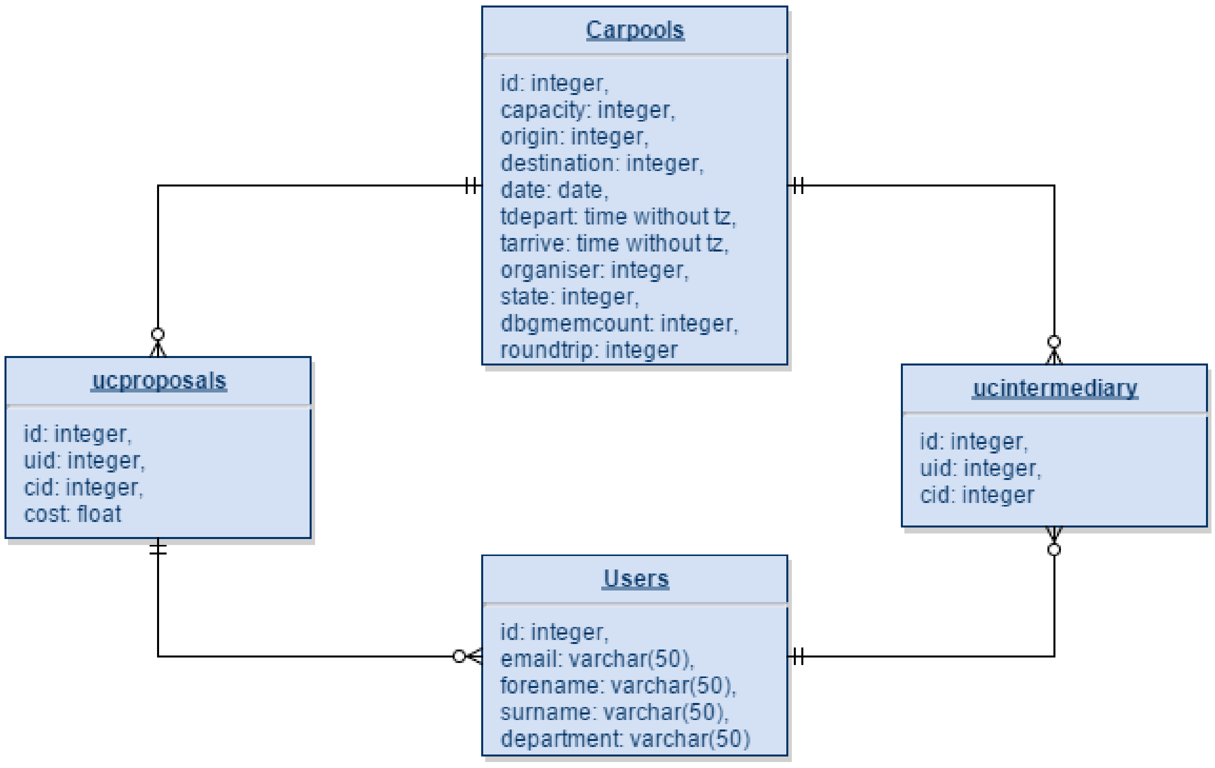


Figure 6: Entity-Relation Diagram

Two NOTIFY triggers are required by the agent system to prevent the need for periodic database polling – the benefits of this are that database listeners will be informed immediately (as opposed to whenever their next poll occurs) and the agents do not have to track their last read row in order to determine whether there has been an insertion since the last check. These triggers are on the Users and Carpools tables, calling a function which sends a NOTIFY to the table's channel containing the row ID to simplify the logic on the agent's end.

4.3. Application Programming Interface

The web API exists to provide a gateway to the backend which is immutable from the client application's perspective. It empowers backend developers to make drastic changes to business logic without requiring client updates, provided that the API endpoints remain the same and the wrapper scripts are updated to correctly bridge the two domains.

In order to meet the performance and availability requirements, some key decisions must factor into the design. The most significant of these is the architectural style of the web API. The system architecture proposed here will benefit from the use of a RESTful architectural style for a number of reasons:

1. REST APIs provide extremely strong support for read caching, making the web service very efficient and minimising the data transfer required on clients which may be charged based on the amount of data transferred over a given period (i.e. mobile data).
2. REST APIs lend themselves to scalable solutions due to the strict forbiddance of conversational state; clients do not care if the server they're talking to is different every time, therefore the web service does not have to track connections in any way. Overall performance is also improved due to the fact that all data required to respond to a request must be included in the request – there are no client-conditional lookups or calculations to be made.

To conform to the RESTful architectural style (Fielding, 2000), the web service must conform to the following criteria:

1. Interaction between client and server must be stateless between requests: there is no notion of ongoing conversation and all data required to execute a request must be provided in the request.
2. Clients can cache responses, therefore the web service should explicitly define responses as cacheable.
3. The system must be layerable, allowing for intermediary systems to be introduced to improve system scalability – for example a load balancer, central cache or web application firewall (WAF).
4. Provide a uniform interface which simplifies and decouples the architecture, through:
    a. Identification of resources in requests (general using URIs in a web API);
    b. Ensuring manipulation of the above resources by clients is possible using only their stored representation of them – the client must be served all required information;

c. Self-descriptive messages which describe to the client how it should process the message, such as media type and cacheability; and
d. Explicitly defined state transition resources as part of a response, before which clients should not attempt state transitions to any resource which is not explicitly defined as a fixed entry point to the web API.

Method	URL	Data	Result
POST	/v1/register	Email Forename Surname User Tags	User registration.
GET	/v1/users		Returns a list of all system users.
GET	/v1/users/<int:user_id>		Returns the public details of a single user.
GET	/v1/carpools		Returns a list of all arranged carpools in the system.
GET	/v1/carpools/<int:user_id>		Returns a list of carpools involving a single user.
GET	/v1/carpools/<int:carpool_id>		Returns details of a single carpool.
POST	/v1/carpools	Capacity Origin Destination Date Departure Time Arrival Time Round Trip	Inserts the details for a new carpool search, settings the organiser as the authenticated user.
GET	/v1/intermediaries		Returns a list of intermediary table entries between users and carpools.
POST	/v1/proposals	UID CID	Marks a proposal as accepted by the authenticated user.
GET	/v1/proposals/<int:user_id>		Returns all proposals which have been made to a single user.

Table 2: API Endpoint Specification

The API entry point design stems from the use cases of the client (which in turn stem from the use cases of the end user). As specified in the requirements analysis, these are:

1. Authenticate
2. View carpool details
3. Initiate carpool searches
4. Accept carpool offers

Which, with regards to the web API, imply a number of subgoals required for clients to provide system users with the required information to interact with the system:

1. Retrieve details of the end user
2. Retrieve a list of intermediary table entries (as specified above, these are used in the database to emulate an array of foreign keys)
3. Retrieve the details of carpools which the end user is a member of
4. Retrieve all proposals which require the end user’s attention

It should be kept in mind that clients may not use all of these endpoints, and indeed a number of endpoints should not be exposed to the end user applications – these are easily identified as the calls which do not return conditional information based on the provided user ID. These endpoints have been included in the specification as they are extremely useful for administration, maintenance and debugging.

As the API will be exposed over the internet to support roaming client applications, security is a critical consideration. To ensure user data is accessible only by the user it relates to, an open standard for client authorisation should be implemented – such as OAuth 1.0 or Security Assertion Markup Language (SAML) 2.0. Importantly, the implementation should enforce a secure communication protocol such as Hypertext Transfer Protocol (HTTP) over Secure Sockets Layer (SSL), commonly referred to as HTTPS. With this enforcement should come secure development decisions to protect the end user such as certificate pinning or end-to-end encryption which prevent man-in-the-middle attacks.

However, securing the client does not secure the API itself. Standards to verify the integrity of a message (that is, to verify that it has not been tampered in transit by an attacker or spoofed using command line tools such as cURL) are well established: key-hash message authentication codes (HMAC) allow messages to be signed using a secret key which is known only to the client application, preventing arbitrary requests from untrusted sources while also verifying message integrity in transit. The server holds the same private key and can use it to recalculate the message hash for verification purposes – and to sign a response which the client can then verify in the same manner. This provides two-way trust and ensures that the server itself is not an easy attack vector.

4.4. Deployment

The scalable and decoupled nature of the architecture, combined with its tendency towards web-facing implementations, lends itself to a cloud computing deployment. Not only does this provide a convenient and hardware-free deployment – but also the ability to deploy globally (perhaps for corner cases such as group airfares), ensure component redundancy, scale automatically in real-time (no lead time on hardware purchasing), and remove large maintenance workloads from the internal teams.

The architecture proposed in this paper will be based upon Amazon Web Services for reasons detailed in Section 2.4. The architecture is deployable on most similar cloud computing providers – such as Google, Microsoft and IBM – as well as on any heterogeneous network whether it is in-house, cloud-based or hybrid.

The terminology used in Amazon Web Services deployment can be confusing at times. For the benefit of understandable deployment on arbitrary infrastructure, a brief explanation will be given here:

- EC2 – Elastic Compute Cloud: Resizable cloud computing instances (virtual machines) which support the booting of machine images to automate the configuration and deployment of applications.
- Route 53: High availability scalable Domain Name System (DNS) which supports modification of records through web service calls, such as adding a new EC2 instance to a hosted zone programmatically when it comes online.
- Elastic IP: Static IP addresses designed to improve availability through programmatic remapping of public IP addresses to instances (for example to mask failures without having to wait for DNS to propagate to clients).
- S3 – Simple Storage Service: Secure, durable and highly-scalable object storage. File containers in S3 are referred to as buckets, with each bucket having a globally unique identifier and being globally distributable.
- ELB – Elastic Load Balancing: Automatic distribution of incoming traffic across EC2 instances, providing high levels of fault tolerance and dynamically scaling load balancing capacity based on traffic levels. ELB also supports auto scaling of instances based on user-defined conditions, using metrics such as average CPU load and minimum latency.
- Availability Zone: An isolated datacentre in a given region – allowing developers to deploy in multiple availability zones per region for higher availability in case of outages.

- Security Group: Virtual firewalls which control traffic for their associated instances.

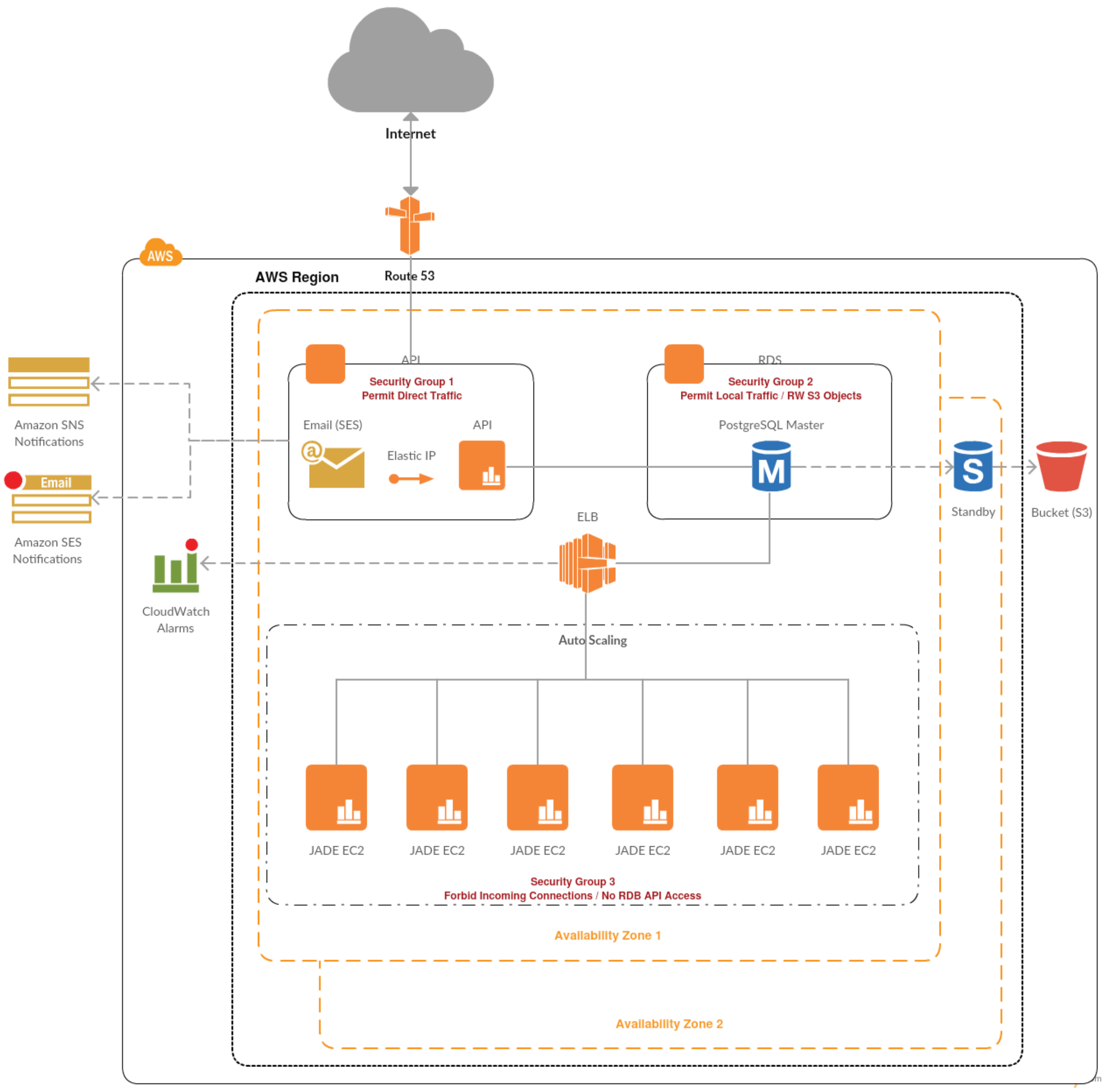


Figure 7: Cloud Deployment Diagram

The ideal deployment of the architecture is in a virtual private cloud, functioning as a private network with none of the backend exposed whatsoever – there is only one path into the network which is the web API itself. From here the traffic may be verified with the integrity checks suggested in Section 4.3, restricting access to the network to trusted clients only. The

API will also handle authorisation checks at this stage to ensure that the user is entitled to access the backend resources which they are requesting, executing the relevant query if successful.

The database itself is mirrored in a separate availability zone to allow quick migration of the system in case of an outage, with periodic database dumps being stored in an S3 bucket to enable disaster recovery (such as database corruptions which are mirrored across availability zones). Database updates are broadcast to all JADE EC2 instances, while new carpool requests will be sent via the load balancer to ensure that the new agent spawns on the least taxed system.

CloudWatch allows the auto scaling cluster of containers to expand and contract as necessary, providing alarms which can be sent to agents on instances which are about to be destroyed or to system administrators when a problem is detected.

4.5. Client Application

Client applications within the system can take almost any form, but in order to best meet the flexibility and usability requirements there are some clear decisions to be made. The two clear types of client application are websites and mobile applications. The design decisions for these are significantly different as the use cases are polar opposites: websites are likely to be accessed from a desktop or laptop on the organisation's private network, implying that the entire system can be hosted on an intranet to remove the majority of security concerns – however this use case does not make much sense as the nature of carpools mean that the system user will likely not have a workstation at one end of the journey. Mobile applications on the other hand allow the user to take the details of their carpools with them, providing a convenient method of double checking the arrangements when with an intranet site they could not. Integrations such as phone numbers (for use cases such as making contact with the carpool members and arranging specifics such as where to meet, or chasing up members who may be running late) make far more sense on mobile platforms, and powerful server-side options such as push notifications become available to make the system far more convenient for end users. For these reasons the architecture design will focus on mobile applications for end users – but this does not mean other client types are not possible. As long as a platform's stack supports, or can be modified to support, RESTful API consumption, the client can be developed to target that platform.

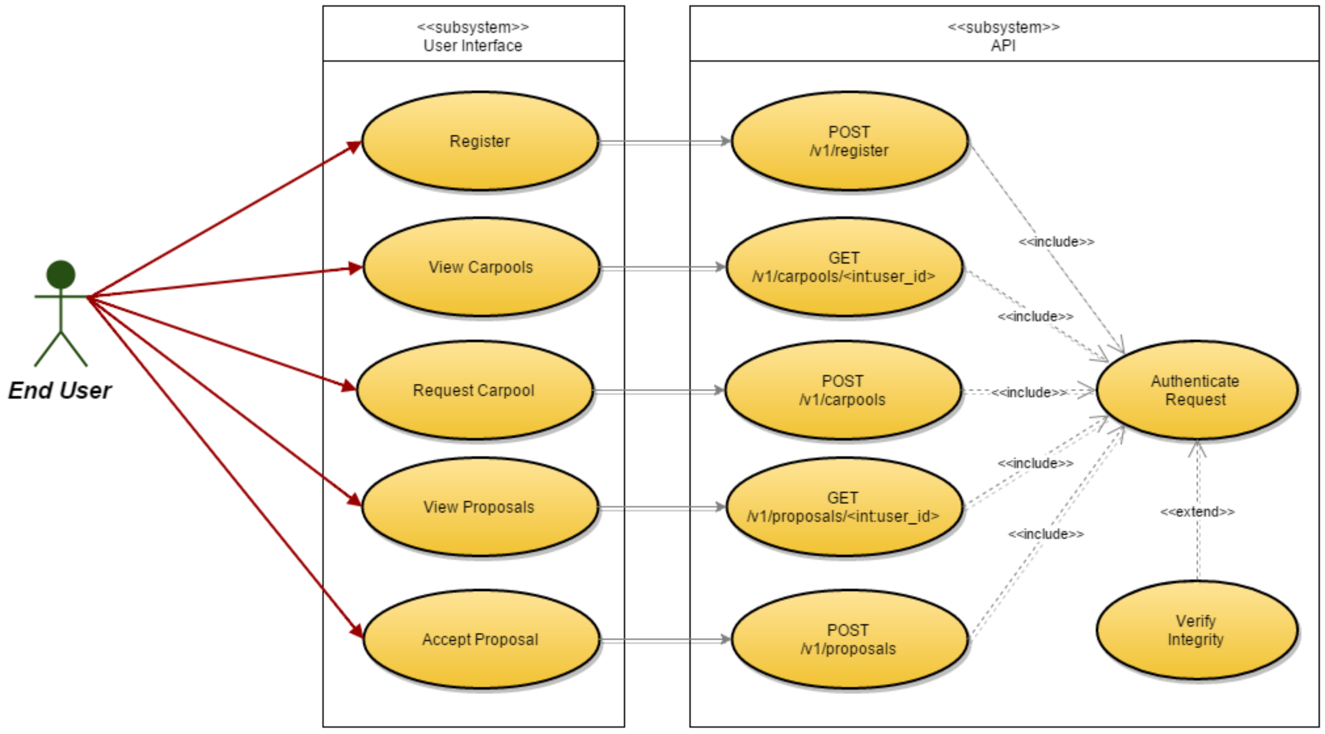


Figure 8: End User Application Use Case

Mobile applications which are for internal organisational use are unlikely to be distributed through application stores therefore the design must provide a method of easily updating all clients without manual intervention on every installed device. This implies the design should favour either HTML5 mobile applications or hybrid mobile applications – both of which are approaches supporting deployment of one implementation across multiple platforms (Android, iOS, Windows Phone, FirefoxOS). However given the usability requirements, hybrid applications clearly stand out as the best choice for the system architecture. Hybrid applications not only allow the application to be updated on the fly by implementing a full web application on the server with the application simply acting as a wrapper for this deployment, but also provide access to native capabilities such as standard system gestures, GPS and much more. Frameworks such as Ionic and Apache Cordova make it extremely easy to develop hybrid applications which retain a strong native focus across supported platforms, allowing convenient access to the native device APIs when required. Both of the example frameworks also support live reloading of deployed apps, making it extremely simple to update the entire installed device base without requiring any user interaction – as well as pseudo app stores to allow private deployment of hybrid applications.

5. Implementation

5.1. Scenario

As a proof of concept implementation of this architecture, a carpooling system tailored for Edinburgh Napier University's requirements will be implemented. The scenario is intentionally simple to stay within the scope and time limits of this project.

Staff require to travel between campuses for meetings, lectures, examinations and other such events. Due to the time constraints of the working day, it is often necessary to commute by taxi – a very high expense to the university when the taxis are below maximum (or at minimum – i.e. one person excluding the taxi driver) occupancy. In a slightly less economically damaging scenario, staff may drive between campuses: this situation is still expensive as fuel costs are multiplied unnecessarily, as well as being environmentally damaging due to multiplied emissions.

The requirements (additional to those defined in Section 3) of the proof of concept scenario are as follows:

- Carpool capacity must be specifiable (to support all kinds of cars, taxis and minibuses);
- There are three locations between which carpooling will occur: Sighthill, Merchiston and Craiglockhart;
- Staff should be able to add user tags – at the very least, their department and role:
    - Arts, Business, Computing, Engineering, Health
    - Lecturer, Senior Lecturer, Reader, Professor
- Staff should be able to specify whether or not their journey is a round trip, as many staff will want to return to their primary campus after their meeting or lecture.

Further user requirements specific to the Napier scenario will not be considered due to scope and time constraints – this proof of concept exists for architecture validation and evaluation rather than as production software for acceptance testing.

5.2. Matching Algorithm

The logic required in the matching algorithm required for Edinburgh Napier University is fairly simple:

- The origin and destination are represented as integers, and for two journeys to be compatible these must both match;
- Time windows are specified as the earliest possible time that a user is willing to commute in the carpool and the latest they are willing to arrive – these must have sufficient overlap to be considered compatible;
- Round trips are separated into two distinct route requests, allowing one-way carpoolers to match with round trip carpoolers and overall increase the chances of a match; and
- Cost is calculated using fixed values provided by TaxiFareFinder (2015) – though this should be replaced with a suitable developer API for a scalable and adaptive solution – and results are presented to the user in descending cost order.

Napier-specific code is required in only two of the RouteAgent states: SEEK_OFFERS and AWAIT_REQUESTS. This tailored code is simply the constraints within which a carpool request is considered viable for integration into the receiving carpool.

```
// Carpool request message
content =  details.getProperty("date")+","
  + details.getProperty("origin")+","
  + details.getProperty("destination")+","
  + details.getProperty("tdepart")+","
  + details.getProperty("tarrive")+","
  + carpoolers.get(0).getLocalName();
```

Figure 9: Tailored Carpool Request Message

As the Napier scenario has three fixed destinations, we can simply assign each an integer value and check that these are the same in both carpools. More complex routing based on a geographic coordinate system or additional parameters is possible simply by extending this code section to accommodate the additional requirements. “content” is a string of arbitrary length and format which can be parsed at the receiving end in any way the developer desires – in this case is it sent as a comma-delimited list which is trivially split on commas into an array of String at the receiving end.

```
1  Time tDepart = Time.valueOf(args[3]);
2  Time tArrive = Time.valueOf(args[4]);
3  Time mArrive = Time.valueOf(details.getProperty("tarrive"));
4  Time mDepart = Time.valueOf(details.getProperty("tdepart"));
5
6  boolean feasible = true;
7
8  String window;
9  earliest = Time.valueOf("00:00:00");
10 latest = Time.valueOf("00:00:00");
11
12 // Determine whether the time windows are feasible
13 // and set the new time window to the largest window
14 // which works for all members
15 if (mArrive.before(tDepart) || mDepart.after(tArrive)) {
16   feasible = false;
17 } else {
18   if (mArrive.before(tArrive)) {
19           latest = mArrive;
20   } else {
21           latest = tArrive;
22   }
23   if (mDepart.before(tDepart)) {
24           earliest = tDepart;
25   } else {
26           earliest = mDepart;
27   }
28 }
29
30 // If origin and destination match, plus time windows are feasible
31 if (details.get("origin").equals(args[1]) &&
32     details.get("destination").equals(args[2])
33     && feasible) {
34   response = new ACLMessage(ACLMessage.PROPOSE);
35   // Work out the new cost per member and set it as the message content
36   TaxiCalculator tc = TaxiCalculator.getInstance();
37   float cost = tc.getCost(details.get("origin") + "," +
38                details.get("destination")).get(1)/(noMembers+1);
39
40   content = cost + "," + earliest.toString() + "," + latest.toString();
41   response.setContent(content);
42
43 // Else we can't carpool with the sender
44 } else {
45   response = new ACLMessage(ACLMessage.REFUSE);
46 }
47
48 // Send the response
49 response.addReceiver(request.getSender());
50 send(response);
```

Figure 10: Tailored Carpool Request Handling

On the receiving end, two checks must be made to meet the Napier requirements for a carpool to be considered viable: both origin and destination much match (Figure 10, Line 38-39), and the time windows must not conflict (Figure 10, Line 22-35). The comparison of origin and destination are extremely trivial due to the scenario, but it is easy to see how a more complicated matching system could be implemented here – in essence, the above code can be extended to support any complexity of matching including mid-route pickups and mid-route drop-offs. The time window calculations however could be considered architecture-specific, but are included here in the interest of systems which may wish to be more lenient: perhaps introducing a tolerance of fifteen minutes at each end of the window or for some reason enforcing precise start and end times – it is worth reiterating here that carpool flexibility was found to be the largest factor for carpool system adoption (Abrahamse and Keall, 2012).

5.3. Reputation

Basic reputation as a proof of concept is included in the system implementation. As detailed in the literature review, at the highest level, reputation is simply a measure of how trustworthy an agent is. If an agent has successfully carpooled with an agent before then it can trust (or not trust) them somewhat reliably based on the quality of the carpool. If there is a mutual agent with which both sides of the negotiation have carpooled with before, then this trust can be extended – though notably the certainty is a lot lower. To stay within the scope and limitations of the project, this primitive form of reputation was introduced to the matching algorithm – specifically to the proposals which are returned to the user, alerting them of first and second degree connections to aid in their decision about which proposal to accept.

This can be conveniently represented as a graph in which nodes are agents, edges are past carpools and the weight of the edge is the number of past carpools between the two nodes. If there is no edge between agents then their proposals are still offered to the user for consideration, as oftentimes in a private system there is a baseline of trust simply due to a shared organisation with restricted entry (such as academia or businesses). Edge weights are only considered in the comparison of agents with the same degree of separation from the carpool seeker, for the purpose of ordering by confidence within these degrees. See the Future Work section for details on the ideal implementation of reputation within the system.

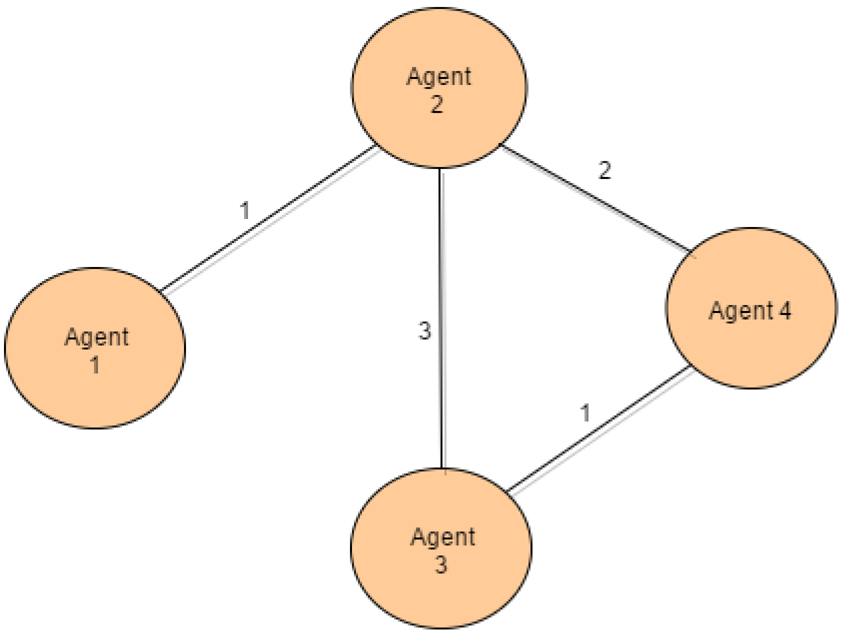


Figure 11: Agent Social Network

From the perspective of Agent 1, the only first degree relation is Agent 2. The two second degree relations are Agent 3 and Agent 4, with Agent 3 taking precedence over Agent 4 due to the higher number of carpools shared with Agent 1's first degree relation(s).

5.4. Deployment

The system is deployed across Amazon EC2 instances and a single Amazon relational database service (RDS) instance. The decision to keep all RouteAgents in a single container initially was made in the interests of determining the exact limits per container given fixed computing resources, with the ability to scale up due to the architectural decisions made in Section 4.4 to enable scalability testing.

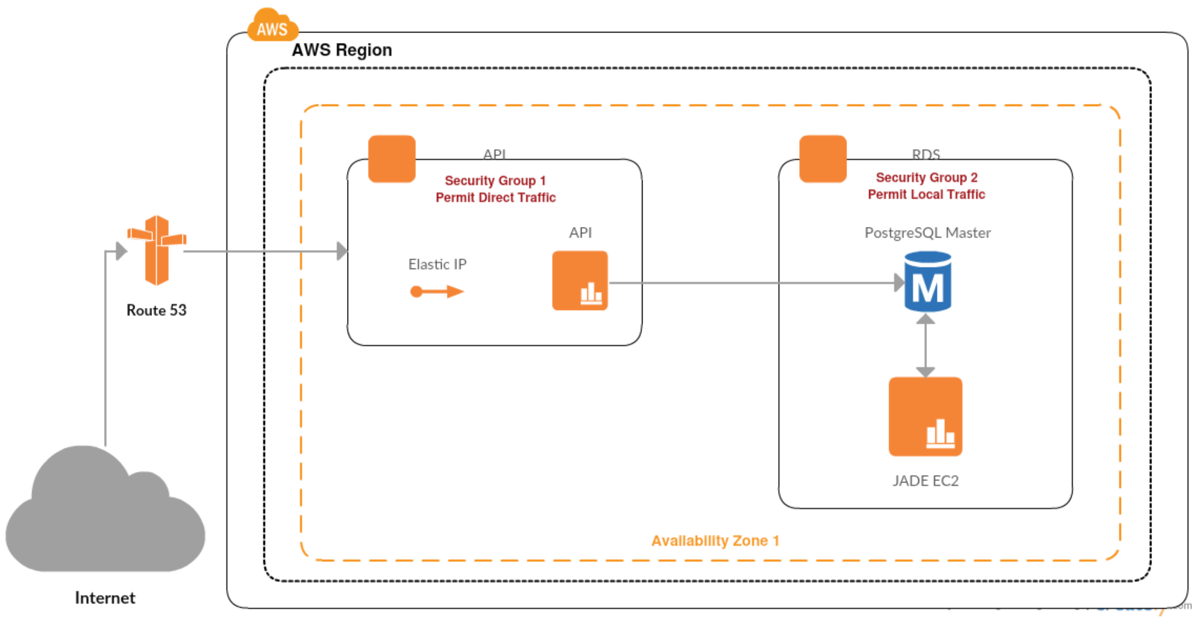


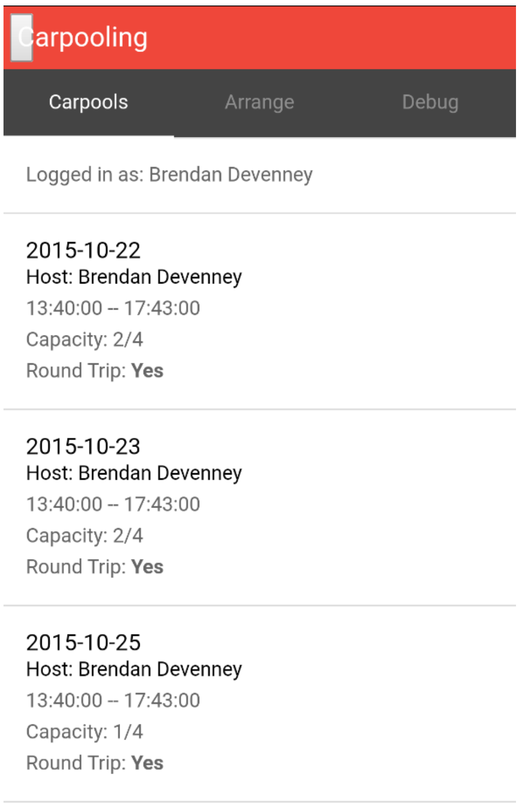
Figure 12: Amazon Web Services Deployment

Comparing this deployment to that specified in the architecture, two things become apparent: many of the user-centric services (email notifications, push notifications, multiple availability zones, and automated database mirroring) are not employed as they are non-core features of the architecture; and a load balancer is not used for auto-scaling in order to retain full control of the system when validating and evaluating.

5.5. Client Application

The proof of concept client application was developed using the Ionic framework (Drifty Co, 2015) – apps are written once in HTML, CSS and AngularJS then deployed to multiple platforms (Android 4.1+, iOS 7+ and BlackBerry 10). The Ionic framework allows rapid prototype development within the scope and time constraints of the project without sacrificing design or user experience. Additionally, developing the application in web technologies has the secondary benefit of proving that the client application is viable as a responsive website. The Ionic framework is built on top of Apache Cordova which wraps these web technologies and extends them to access the native benefits of the target platform. This means features such as native scrolling and gestures – as well as explicitly defined native functions which can be called from the higher code layer.

It should be noted again, as stated in Section 5.1, that user requirements specific to the Napier scenario – for example, style sheets and non-core functionality – will not be implemented or validated. The client application exists purely as a convenient demonstration of the underlying architecture from a user's perspective.



Logged in as: Brendan Devenney

2015-10-22
Host: Brendan Devenney
13:40:00 -- 17:43:00
Capacity: 2/4
Round Trip: **Yes**

2015-10-23
Host: Brendan Devenney
13:40:00 -- 17:43:00
Capacity: 2/4
Round Trip: **Yes**

2015-10-25
Host: Brendan Devenney
13:40:00 -- 17:43:00
Capacity: 1/4
Round Trip: **Yes**

Figure 13: Carpool Details View

As shown in Figure 13, carpool details can be fetched from the API and displayed in the most convenient manner for the organisation's users. In this implementation, the carpools are ordered by date with the organiser (a carpool requester who failed to find an acceptable match and went on to form their own carpool which considers incoming requests) displayed as the member primarily responsible for the carpool. There could equally be a list of members stated here with no clear owner, providing a less structured carpool experience but perhaps more suited to an organisation's culture. The time window which has been calculated as suitable for all carpool members is displayed as well as information on whether or not the carpool is a round trip.

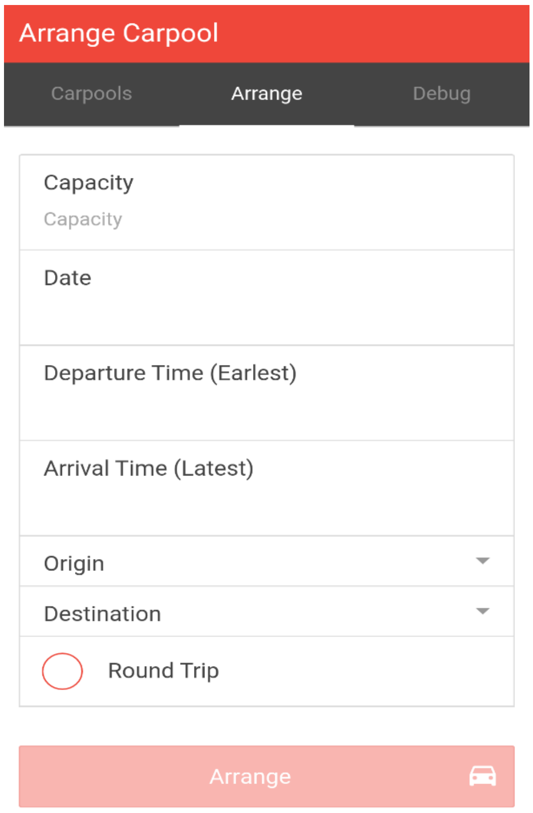
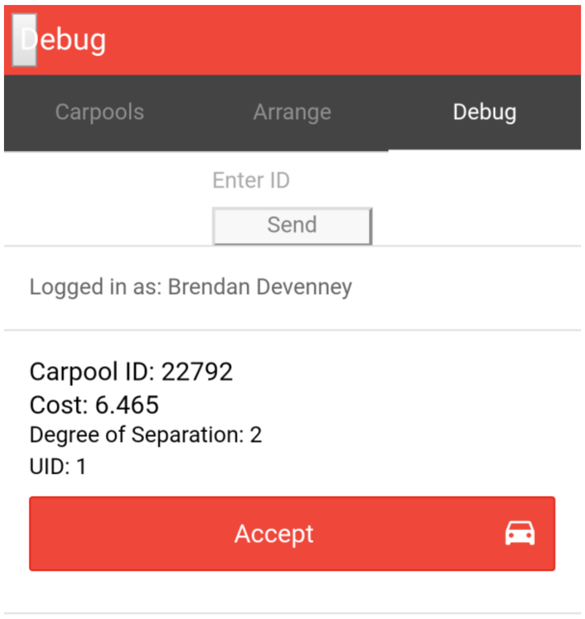


Figure 14: Carpool Arrangement View

Users initiating carpools is simply handled as a form. The user may specify the capacity, date, earliest departure time and latest arrival time. The origin and destination are chosen from a list of the three Napier campuses which are hard coded due to project constraints, but could be pulled from the API in a production system which would allow the application to update with new venues instantly. When the arrange button is pressed, the user's agent will silently spawn and begin negotiating with suitable agents to find the best proposals to offer the user. The form is validated using AngularJS and the user is guided in their inputs using HTML5 input types, such as date pickers and input restriction.

The returned proposals are ordered by cost – calculated as cost per member, therefore the maximum that the seeker will have to pay but notably not the minimum – as an organisation such as Napier has a high level of default trust between system users. As demonstrated in Figure 15, the displayed proposals include the degree of separation between the seeker and the offeror – in this case 2, as the users have a mutual past carpool member.



Logged in as: Brendan Devenney

Carpool ID: 22792
Cost: 6.465
Degree of Separation: 2
UID: 1

Figure 15: Carpool Offers View

6. Evaluation and Refinements

6.1. Stress Testing

1. Methodology

Stress testing will consist of three tests: flood, trickle and simulation. The intention of the flood test is to simulate the system's ability to handle an extreme number of carpool requests at a single time, while the trickle test simulates a high number of requests with equal spacing to determine the system's capabilities for dealing with consistent carpool requests. Both of these provide valuable metrics for validation of the efficiency, response time and scalability requirements defined in Section 3. The third test, simulation, is a combination of the first two with an element of randomness introduced to simulate real system use – a randomly spaced trickle of request floods. This is intended to validate the backend of the system against the majority of non-functional requirements – efficiency, performance, response time, and scalability – under realistic load. This test is dependent on the successful completion of the flood and trickle tests, meaning that the system must not fall over as a result of either.

Test	Number of Requests	Interval
Flood	500	0ms
Trickle	1000	5000ms
Simulation	Infinite Clusters (1-10 Requests) Until Failure	0ms-5000ms

Table 3: Stress Test Configurations

The system is prepopulated with only four user agents which are selected randomly for each request, with the date randomly selected from the month of February (2015). The time is randomly selected from AM (09:00-12:00) and PM (12:00-17:00). The capacity is set to two or three. The stress should be increased by the low number of user agents, as it is highly likely that a request for any given day will be incompatible with the pre-existing carpools as a user may not carpool with themselves.

Given the two to three capacity constraint, in the best case scenario it is expected to see a carpool count which is between 50% and 66% of the request count. Due to the random nature of the stress test it is unlikely that the conditions will align to allow this, therefore a carpool count which is slightly worse than 50% of the requests count still strongly suggests the system is negotiating all matches as expected – while a number which is significantly worse than the 50% figure may suggest negotiation has failed for a number of requests, requiring further investigation into the quality consequences of the stress test.

All stress testing was conducted on an Amazon t2.micro instance with the following specification: 1 burstable vCPU (Intel Xeon with Turbo up to 3.3GHz), 1GB RAM. It is noteworthy that this instance is not powerful and is not intended for consistently high CPU usage workloads.

2. Initial Test

The initial flood test resulted in a system collapse due to agents being unable to search the directory facilitator service (DF Service) for potential matches, caused by "*jade.domain.FIPAException: Timeout searching for data into df.*" A quick inspection of the JADE API shows that this exception is thrown when the DFService takes upwards of 30000ms to respond to a search request. This occurred after 67 requests, 23 of which were matched and merged into another carpool – giving an active agent count of 44. This is an objectively miniscule number of agents considering the potential applications of a carpooling architecture. However the results do show one strong correlation: the time to negotiate a carpool increases drastically as the number of active agents involved in the communication increases, directly caused by the DFService taking longer to respond to search queries. This agrees with the findings of Mengistu et al. (2008), in which it was claimed that the JADE agent directory service is inefficient and does not function acceptably in large-scale agent systems with a high rate of communication.

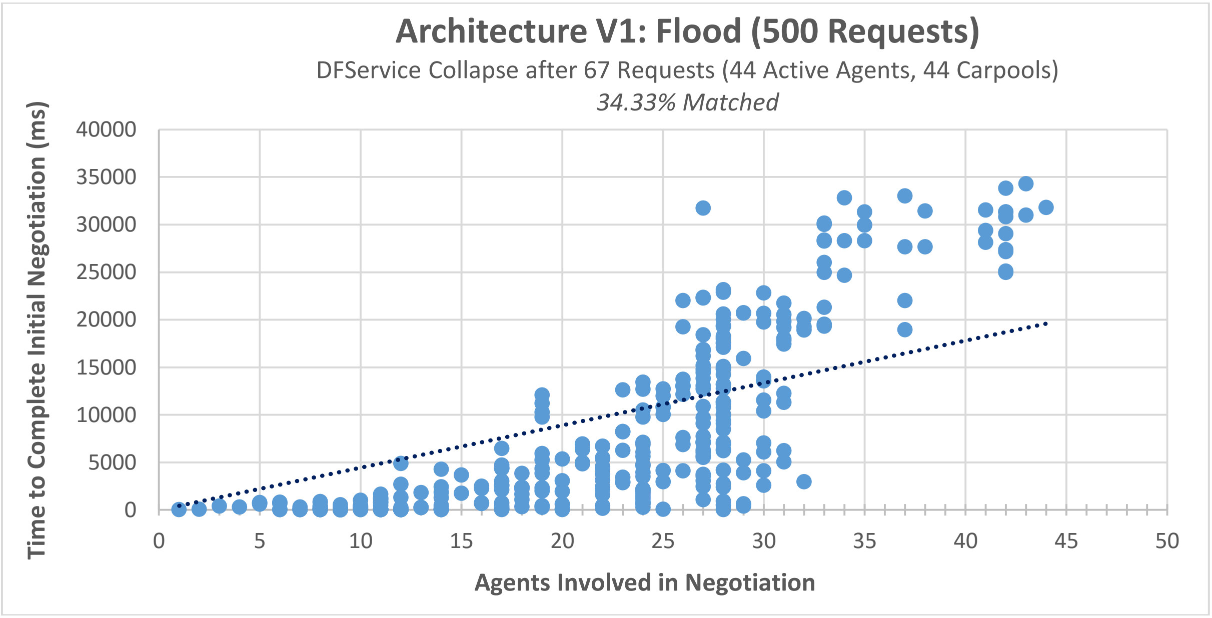


Figure 16: Initial Flood Test Results

The trickle test was slightly more promising but ultimately resulted in the same system collapse after 642 requests and 31 merges, with an active agent count of 286. While this looks better with regards to the number of requests handled before falling over, there is a clear divergence between the number of requests and the number of agents which were spawned. This is due to the agent system failing to process database triggers in a timely manner – so many agents were stuck actively waiting for the DFService to reply that there were insufficient computing resources available for the Agent Factory and Agent Management Service (AMS) to spawn agents in a timely manner. Fairly similar to the flood test, the system handled the requests fairly well for a short time before the DFService response times became unacceptable due to the extreme load placed on it. From a user's perspective this would appear as though they had sent a carpool request and there was no backend system present to handle it, with no proposals ever being returned – entirely unacceptable and can be regarded as a 'soft' system collapse as the functionality had ceased.

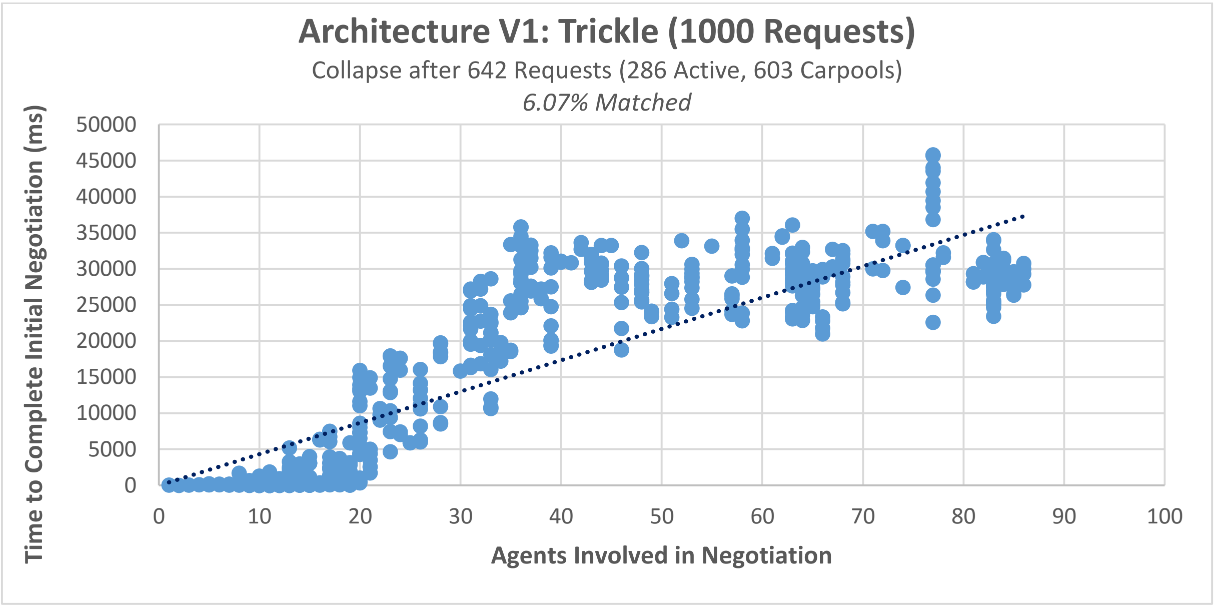


Figure 17: Initial Trickle Test Results

The conclusions of this initial test are as follows:

- The DFService is a finite resource which struggles with a high frequency of searches for service names which have a large amount of registered agents;
- The initial implementation of the architecture cannot handle being flooded at all, though fares slightly better when requests are trickled – suggesting that the DFService benefits from the system having time to settle between requests.

3. Medial Test

In order to tackle the DFService instability, modifications to limit the size of the DFService search were necessary. It was decided that there were two feasible modifications to the advertised service name which would exclusively remove infeasible carpools from the search results: inclusion of the date in the service name and inclusion of the endpoints in the service name. However, inclusion of start and endpoint information is clearly very restrictive – this would not be feasible in systems using a geographic coordinate – therefore it was not included in the medial test. The exact code change is as follows:

```
//sd.setType("carpool");
sd.setType("carpool" + details.get("date"));
```

Figure 18: ServiceDescription Code Comparison

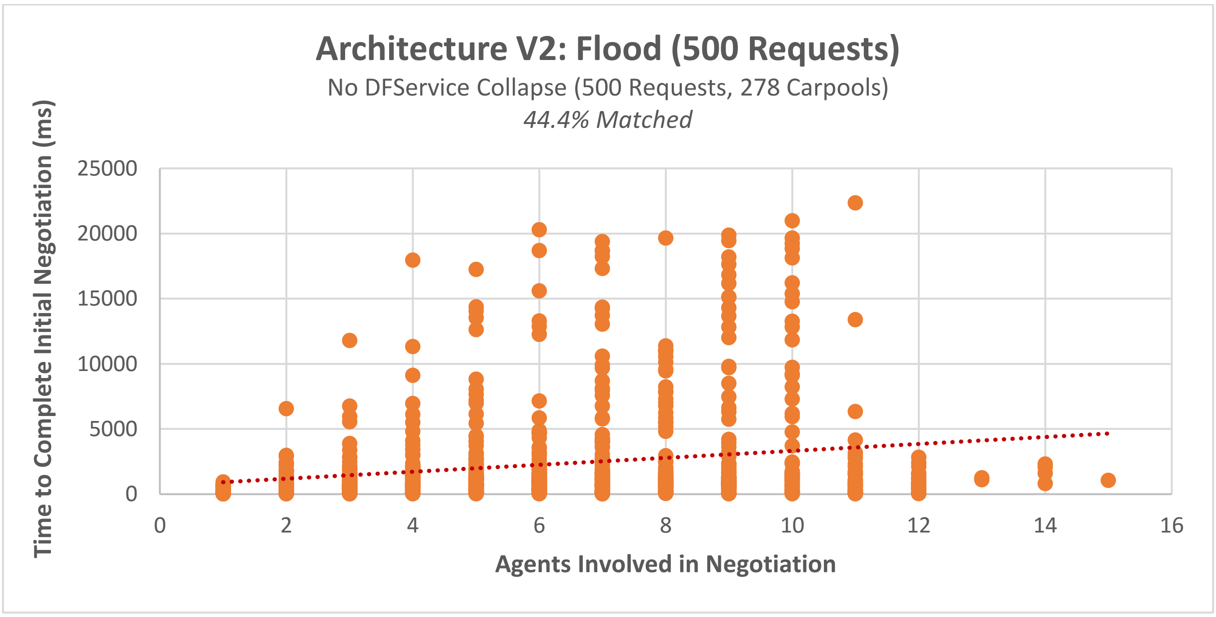


Figure 19: Medial Flood Test Results

This version of the implementation handled the flood test significantly better – there was no system collapse as the DFService had significantly smaller searches to execute. Though the variation in response time was extremely high (ranging from negligible to around 22500ms) it should be noted that the majority of responses returned in less than 15000ms, which is half of what the JADE by default considers to be entirely unacceptable. However the trend line in Figure 19 clearly shows that the response time trends upwards as the number of agents in a search increases which is still a cause for concern given that the DFService underpins all carpool negotiation. Clearly it is a finite resource which will ultimately fail if the system is flooded as heavily as possible for a sustained period of time.

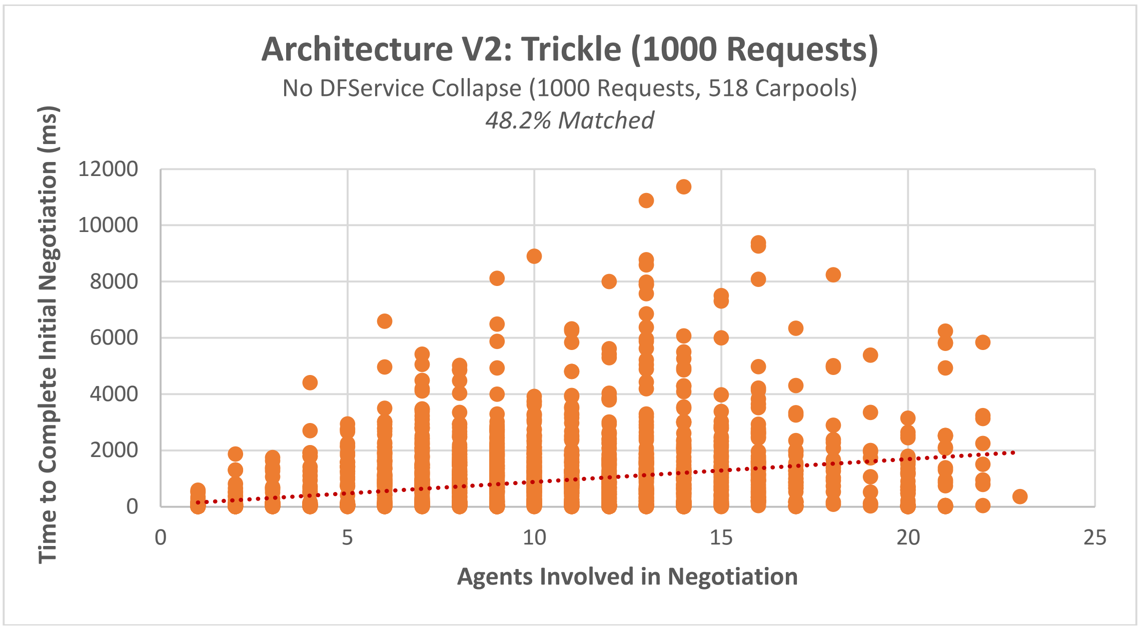


Figure 20: Medial Trickle Test Results

As before, the system predictably handles the trickle test better than the flood request, however the difference is far more extreme in this case. As shown by the trend line in Figure 20, with a constant and consistently spaced stream of requests, the system almost maintains a steady average response time regardless of the number of active agents. Having only two response times above one third of the response time that the JADE platform considers acceptable is reassuring and shows a significant improvement over the initial implementation. However, there is still a trend towards higher response times as the number of agents involved in the communication increases which of course implies system failure in extreme circumstances.

It should be noted that these results are not representative of the worst case scenario in which every carpool in the system has the same date. In such a scenario, the addition of the date in the Service Description is redundant as every carpool's Service will be registered with the same name – essentially mimicking the circumstances of the initial tests. However the chances of this happening in a production system are extremely small (though undoubtedly possible for popular days) and the modification, in general, improves system stability significantly.

4. Comparison

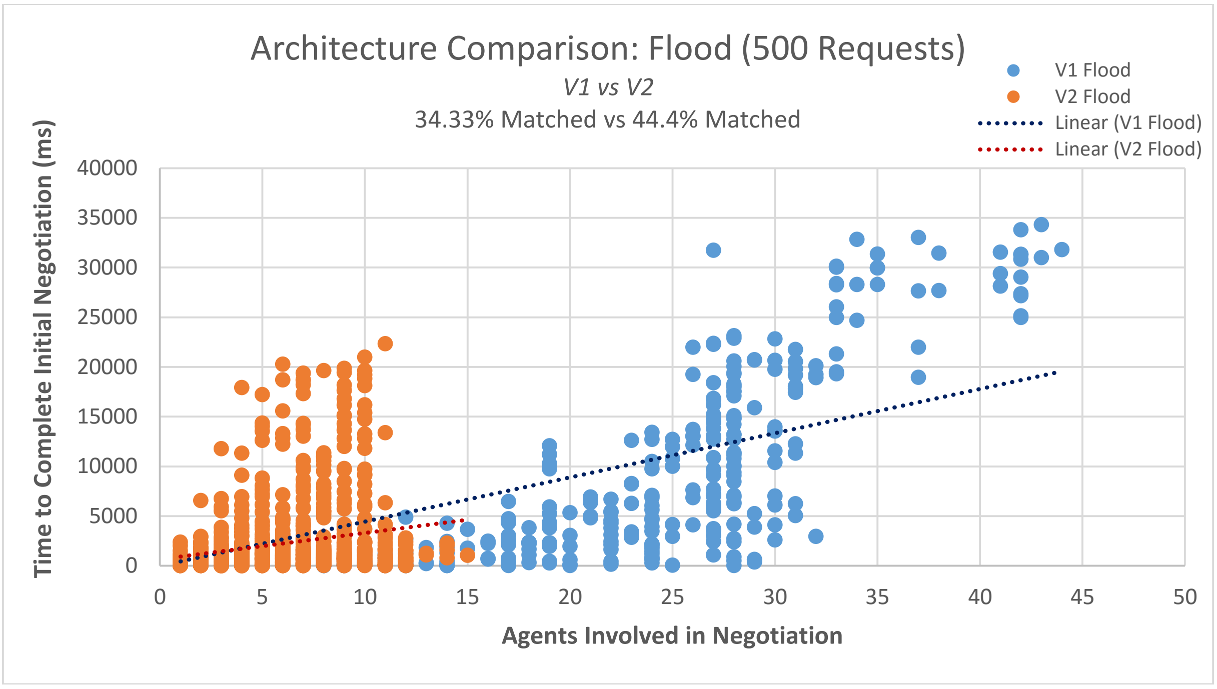


Figure 21: Flood Test Comparison

Comparing the results of the initial and medial tests gives a deeper insight into the system improvements. Firstly, the reduction of the population of agents registered with a given Service Description by including the date in the service name significantly improves the efficiency and stability of the system – agents no longer have to check whether or not the requesting agent is attempting to carpool on the same date as it can be safely assumed, and the requesting agent only has to communicate with a subset of the system's agents. This change also empowers more advanced deployment which improves the scalability of the DFService:

- One DFService per Service Description is possible, allowing developers to have a large number of independent DFService agents federated by a central agent.
- One container per date is also possible, allowing agent communication to be contained within a single container even in distributed systems. This brings with it the obvious benefit of avoiding latency when containers are geographically distributed.

The system appears capable of handling significant floods of requests as well as unrealistic sustained load in the latter tests, in both cases maintaining acceptable responsiveness and correctness of the carpool negotiation algorithm. Though improved significantly, the results still show that the DFService is a finite resource which can be overloaded – a solution to allow the DFService agent to stabilise in extreme circumstances would be a huge improvement to the architecture.

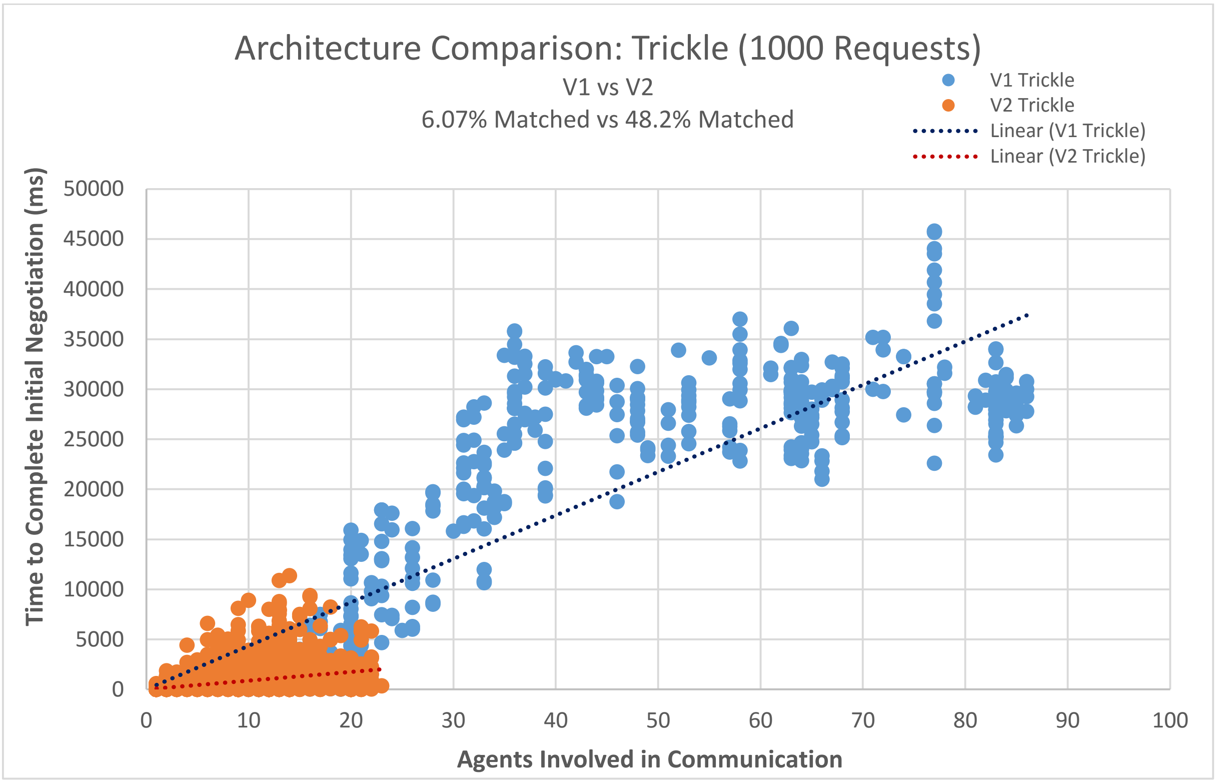


Figure 22: Trickle Test Comparison

5. Final Test

In order to tackle the issue of the DFService agent being a finite resource, an injectable monitoring agent was developed. This agent periodically queries the DFService for all registered agents in order to determine the DFService Agent's current maximum response time. The monitoring agent is supported by a singleton which stores the current status of the DFService:

- OK – Response time less than 3000ms
- SLOW – Response time between 3000ms and 20000ms
- UNSTABLE – Response time greater than 20000ms

Route Agents get the current DFService status from the aforementioned singleton before sending any queries to the DFService. If the status is *OK*, the query proceeds normally; if the status is *slow*, there is a significant risk of the DFService becoming *unstable* before the next response time check – so Route Agents increase their search timeout to avoid entering an expensive query-exception loop; and if the status is *unstable* then the Route Agent will sleep before checking the singleton's stored status again, allowing the DFService to deal with its current load and stabilise.

To verify that the monitoring agent truly helps the DFService avoid instability, a simulation stress test was run until system failure (due to the intentional exclusion of elastic computing, this equates to java heap exhaustion or host system hang – or in the worst case due to the issues observed in the initial and medial tests). Successful monitoring should help the system maintain a response time below the *unstable* threshold – in this implementation, 20000ms. The system should also avoid JADE timeout exceptions by informing agents to increase their timeout before the exception-throwing query would occur.

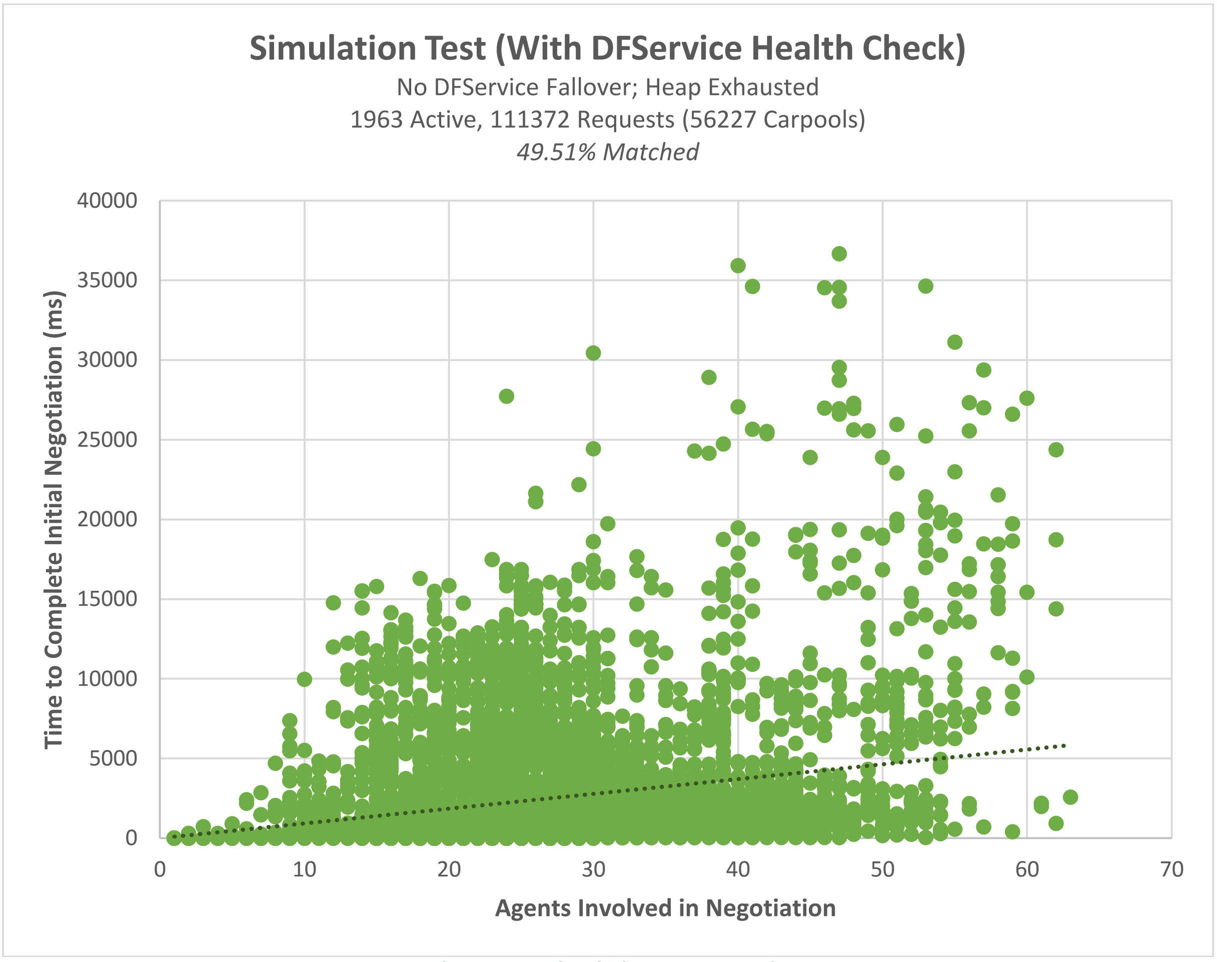


Figure 23: Simulation Test Results

As shown in Figure 23, the results of the simulation test with the added DFService Monitoring Agent are extremely promising. Despite subjecting the system to relentless load for many hours, the system collapse was ultimately caused by the exhaustion of Java heap space which prevented further agents from spawning in the container. Having handled the negotiation of 56227 carpools by the time of system collapse – on a fairly weak system – it seems fair to say that the scalability of the system has been drastically improved from the initial architecture which collapsed after less than one hundred requests.

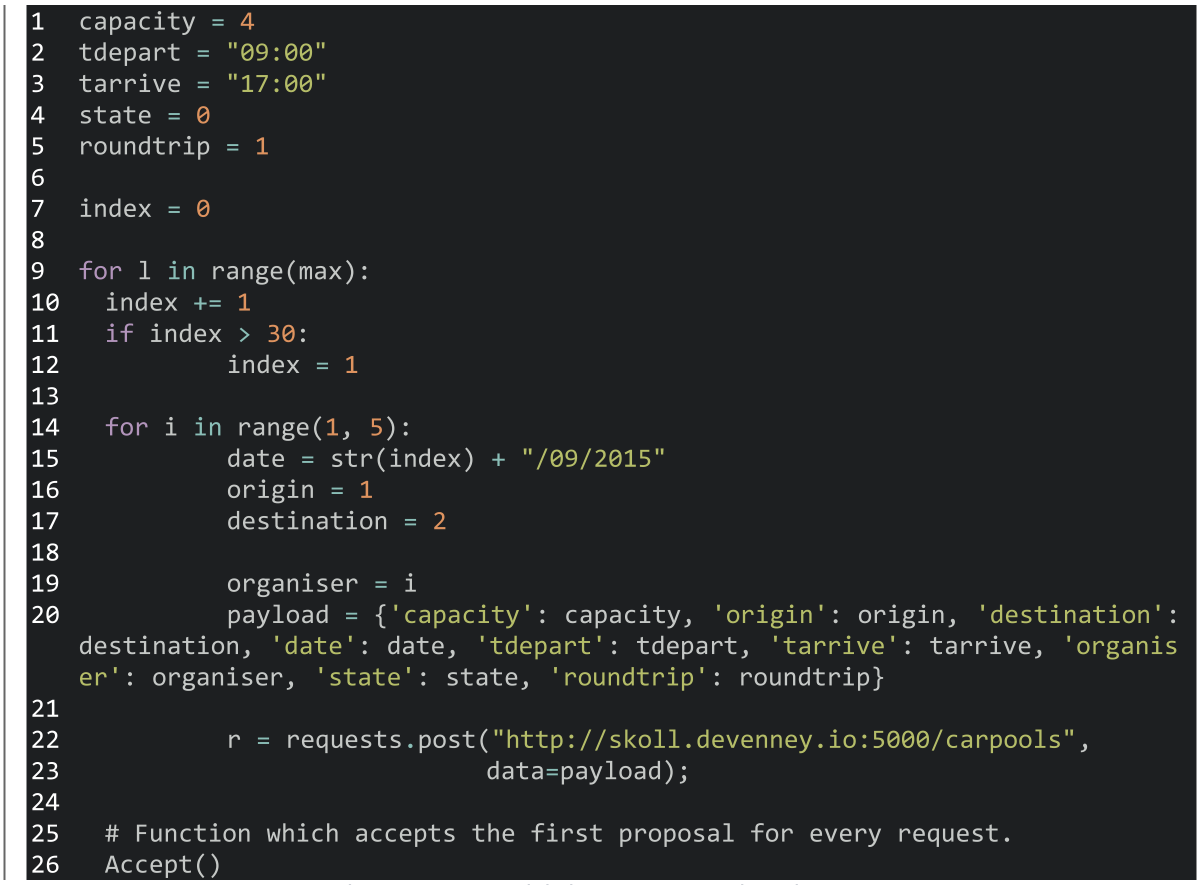
The matched percentage is exactly what should be expected from a correctly functioning system after the sample size of requests has reached a sufficient size – suggesting that the system continued to negotiate all valid carpools despite the mediation of the DFService Monitoring Agent. Perhaps most importantly this proves that the Monitoring Agent did not flag the DFService as unstable without it being able to recover at any point, hence proving that forcing agents to back off temporarily does indeed enable the DFService to stabilise.

The total number of exceptions thrown in the *SEEK_OFFERS* state was zero for this test, while the results contained nine response times which would have been considered exceptions without the DFService monitor (meaning the DFService became *unstable* between the Monitoring Agent’s check and the Route Agent’s query). Given that the exception previously meant that the agent was unable to continue to the next phase of carpool negotiation, avoiding the exception ensures that the end user does not experience a major delay in receiving proposals or a complete lack of system response.

6.2. Matching Algorithm Validation

1. Methodology

It is important to validate the matching algorithm to ensure that matches in the system are accurate – both in terms of bugs which cause invalid matches and performance issues which result in two compatible agents not matching. This validation will be conducted through the sending of predefined carpool requests which have been designed to match. These of course could be matching incorrectly so simply inspecting the number of requests versus the number of matches is inadequate – therefore a handful of established carpools should be inspected to ensure that all of the matches are compatible.



```
capacity = 4
tdepart = "09:00"
tarrive = "17:00"
state = 0
roundtrip = 1

index = 0

for l in range(max):
 index += 1
 if index > 30:
     index = 1

 for i in range(1, 5):
     date = str(index) + "/09/2015"
     origin = 1
     destination = 2

     organiser = i
     payload = {'capacity': capacity, 'origin': origin, 'destination': destination, 'date': date, 'tdepart': tdepart, 'tarrive': tarrive, 'organiser': organiser, 'state': state, 'roundtrip': roundtrip}

     r = requests.post("http://skoll.devenney.io:5000/carpools",
              data=payload);

 # Function which accepts the first proposal for every request.
 Accept()
```

Figure 24: Deterministic Requests Code Snippet

The requests are sent using a Python script – a snippet of which is shown in Figure 24 – designed to be easily scalable without introducing stochastic matching to the validation. Each loop of the script will send four identical requests for an incremented day of the month, for users with ID one to four. These four requests will *always* be compatible and the script will not send another compatible request for 29 iterations of the loop, leaving more than enough time for the carpool negotiation to complete and avoid any potential conflict. There also exists a function which emulates a user's response to offers, which simply emulates the API call of a user accepting the first proposal. Due to the script's design, the user will only ever have one proposal to accept.

Results will consist of executing a *COUNT(*)* query on the *carpools* and *ucintermediary* tables, with the expected results to be an intermediary entries count exactly four times greater than the carpool count – meaning every batch of four requests has negotiated successfully and merged into a full carpool. The second component of the validation will be to inspect randomly selected carpools to ensure the matches are truly valid. For this validation, 80 requests will be sent and five carpools will be inspected.

2. Results

The database queries show that the expected number of successful negotiations took place, with the carpool count being one quarter of the intermediary count. This validates, at the very least, that carpool agents do not exceed the capacity defined in the requests and successfully deregister after negotiating their final spot. It also validates that carpool agents do not get stuck in any infinite loops which cause them to never complete negotiation.

```
carpooling=> SELECT COUNT(*) FROM carpools;
 count
-------
    20
(1 row)
```

Figure 25: Carpool COUNT(*) Query

```
carpooling=> SELECT COUNT(*) FROM ucintermediary;
 count
-------
    80
(1 row)
```

Figure 26: Intermediary COUNT(*) Query

The inspection of random queries within these carpools showed that the matches were valid. In all five randomly selected carpools, the contents were four requests sent in a single iteration of the validation script's loop.

```
{'id': 5, 'organiser': 1, 'tdepart': '09:00', 'tarrive': '17:00',
'state': 0, 'roundtrip': 1, 'origin': 1, 'capacity': 4, 'destination':
2, 'date': '2/09/2015'}
{'id': 6, 'organiser': 2, 'tdepart': '09:00', 'tarrive': '17:00',
'state': 0, 'roundtrip': 1, 'origin': 1, 'capacity': 4, 'destination':
2, 'date': '2/09/2015'}
{'id': 7, 'organiser': 3, 'tdepart': '09:00', 'tarrive': '17:00',
'state': 0, 'roundtrip': 1, 'origin': 1, 'capacity': 4, 'destination':
2, 'date': '2/09/2015'}
{'id': 8, 'organiser': 4, 'tdepart': '09:00', 'tarrive': '17:00',
'state': 0, 'roundtrip': 1, 'origin': 1, 'capacity': 4, 'destination':
2, 'date': '2/09/2015'}
```

Figure 27: Carpool Requests Composition

7. Final Architecture

The final architecture differs from the initial design in several key ways:

- The DFService Agent should ideally be located in a different container than the general carpooling agents in order to prevent carpool negotiation from depriving the DFService Agent of processor time;
- Route Agents should include the desired date when advertising their service, significantly reducing both the load on the DFService Agent and the number of recipient agents which have to reject a carpool request (including a date comparison per recipient) which could never have been compatible in the first instance;
- In an ideal, highly scalable solution there will exist one container per date – a process easily handled by the Agent Factory which can spawn containers if they do not exist and specify the host container in each agent creation request; and
- An agent dedicated to monitoring the DFService should exist which is responsible for keeping track of its status in a singleton. This singleton should be checked by Route Agents prior to beginning carpool negotiation – backing off and sleeping their thread for a few seconds before retrying.

7.1. Agent System

The agent system differs from the initial design in two key ways:

1. *There exists a DFService Monitoring agent which maintains the status of the DFService (OK, SLOW or UNSTABLE) in a singleton accessible by Route Agents.*
2. *Route Agents must check the status held in this singleton before initiating DFService searches, sleeping if the status is UNSTABLE in order to restore system stability.*

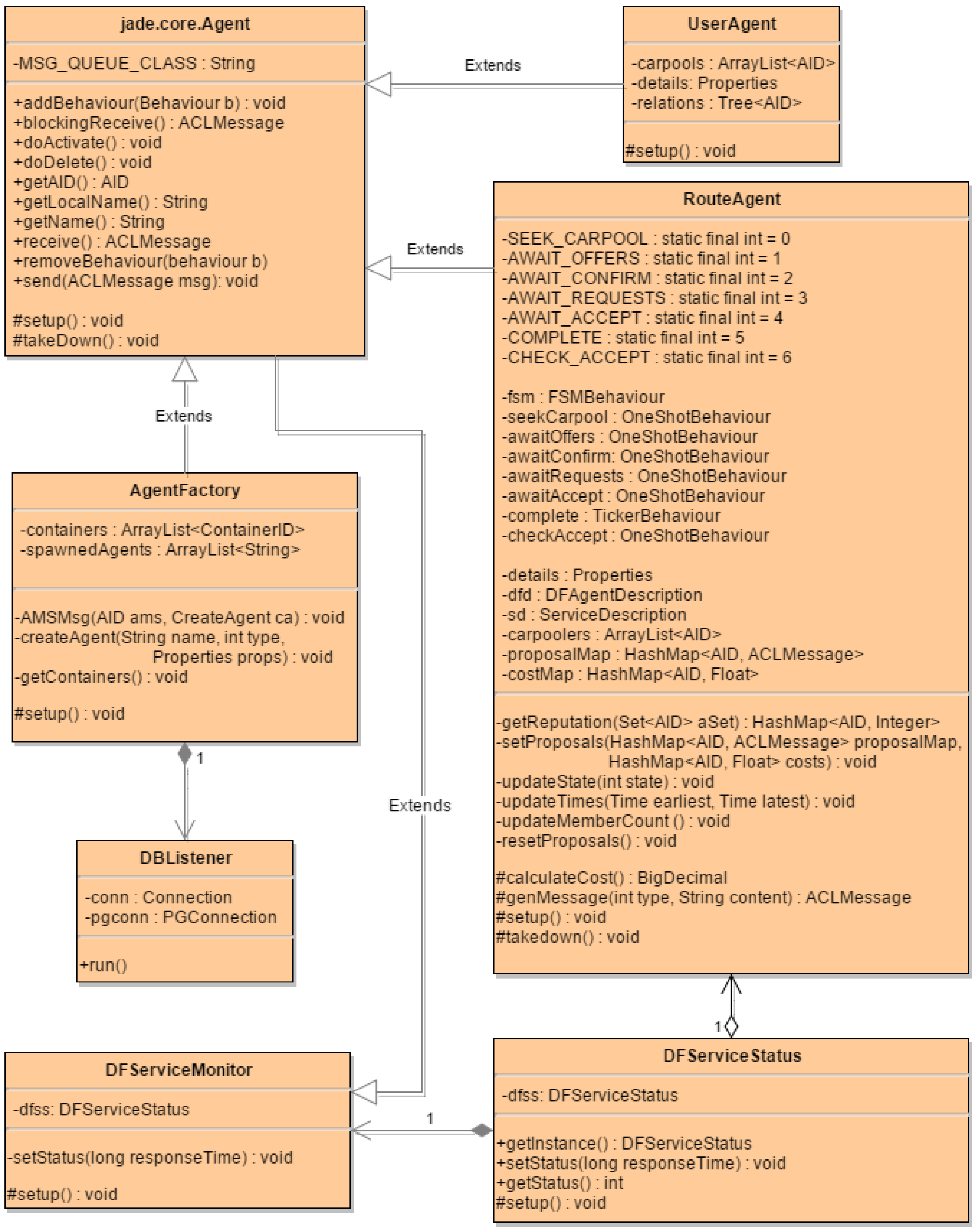

jade.core.Agent
-MSG_QUEUE_CLASS : String
+addBehaviour(Behaviour b) : void
+blockingReceive() : ACLMessage
+doActivate() : void
+doDelete() : void
+getAID() : AID
+getLocalName() : String
+getName() : String
+receive() : ACLMessage
+removeBehaviour(behaviour b)
+send(ACLMessage msg): void
#setup() : void
#takeDown() : void
UserAgent
-carpools : ArrayList<AID>
-details: Properties
-relations : Tree<AID>
#setup() : void
Extends
Extends
RouteAgent
-SEEK_CARPOOL : static final int = 0
-AWAIT_OFFERS : static final int = 1
-AWAIT_CONFIRM : static final int = 2
-AWAIT_REQUESTS : static final int = 3
-AWAIT_ACCEPT : static final int = 4
-COMPLETE : static final int = 5
-CHECK_ACCEPT : static final int = 6
-fsm : FSMBehaviour
-seekCarpool : OneShotBehaviour
-awaitOffers : OneShotBehaviour
-awaitConfirm: OneShotBehaviour
-awaitRequests : OneShotBehaviour
-awaitAccept : OneShotBehaviour
-complete : TickerBehaviour
-checkAccept : OneShotBehaviour
-details : Properties
-dfd : DFAgentDescription
-sd : ServiceDescription
-carpoolers : ArrayList<AID>
-proposalMap : HashMap<AID, ACLMessage>
-costMap : HashMap<AID, Float>
-getReputation(Set<AID> aSet) : HashMap<AID, Integer>
-setProposals(HashMap<AID, ACLMessage> proposalMap, HashMap<AID, Float> costs) : void
-updateState(int state) : void
-updateTimes(Time earliest, Time latest) : void
-updateMemberCount () : void
-resetProposals() : void
#calculateCost() : BigDecimal
#genMessage(int type, String content) : ACLMessage
#setup() : void
#takedown() : void
Extends
AgentFactory
-containers : ArrayList<ContainerID>
-spawnedAgents : ArrayList<String>
-AMSMsg(AID ams, CreateAgent ca) : void
-createAgent(String name, int type, Properties props) : void
-getContainers() : void
#setup() : void
1
Extends
DBListener
-conn : Connection
-pgconn : PGConnection
+run()
1
DFServiceStatus
-dfss: DFServiceStatus
+getInstance() : DFServiceStatus
+setStatus(long responseTime) : void
+getStatus() : int
#setup() : void
DFServiceMonitor
-dfss: DFServiceStatus
-setStatus(long responseTime) : void
#setup() : void
1

Figure 28: Final Agent System Class Diagram

The final agent system consists of three core agents and two optionally injectable agents. The core agents remain concerned with the real-time carpool negotiation, with the relevant changes being: the way in which the agents register their service now includes the desired date of the carpool, and the inclusion of a DFService Status singleton which is queried to determine an agent's ability to search based on the DFService Status. The new injectable agent is concerned with monitoring the health of the DFService and imposing restrictions on agents to maintain stability (useful when the system demand is high but computational resource availability is low).

The three core agents are as follows:

1. The Factory Agent is responsible for the construction and delivery of agents.
    - Creation messages to the Agent Management System (AMS) agent are used to programmatically spawn agents.
    - On inception, data about all users and carpools is pulled from the database and used it to respawn any agents which existed during a previous run (generally in the case of a system restart, though this method can theoretically handle a database from another architecture to migrate from a non-agent system to the architecture proposed in this paper).
    - There exists a PostgreSQL database listener which listens for insertion and deletion notifications on the user and carpool tables' channels, spawning and destroying agents to synchronise the environment.
2. The User Agent represents a single user.
    - User Agents hold a two-deep tree of related User Agents (direct relations which have formed successful carpools with their related user, and indirect relations which have formed successful carpools with the direct relations) to provide basic reputation within the system.
    - User Agents are also responsible for the removal of its user from all relevant carpools when the user is removed from the system.
3. The Route Agent represents a single carpool.
    - Route Agents communicate between themselves to autonomously retrieve the best potential carpools to present to the end user based on their requirements, ultimately merging together as a result of the end user's acceptance of an offer.
    - On its inception, it must populate a key-value list of all carpool properties passed to it as part of the agent creation message. These values may be as simple or complex as the implementation requires.

- Agent behaviour is dictated by a finite state machine (represented in JADE as an FSMBehaviour which holds states, transitions and transition values). In general there are three states in which agents are not awaiting replies, therefore there are three states in which agents may be restored: SEEK_OFFERS, AWAIT_REQUESTS and COMPLETE. The initial state of the finite state machine is LOAD (not documented in the diagram for the sake of clarity) which may transition into any of the aforementioned based on the state which is held in the database entry. New carpool requests will set this to SEEK_OFFERS by default.
- Before initiating an agent directory search, the agent must extend timeouts or sleep based on the DFService status.

State	ID
SEEK_OFFERS	1
AWAIT_REQUESTS	2
AWAIT_ACCEPT	3
ACCEPT_OFFER	4
AWAIT_CONFIRM	5
COMPLETE	6

Table 4: Final Agent System States

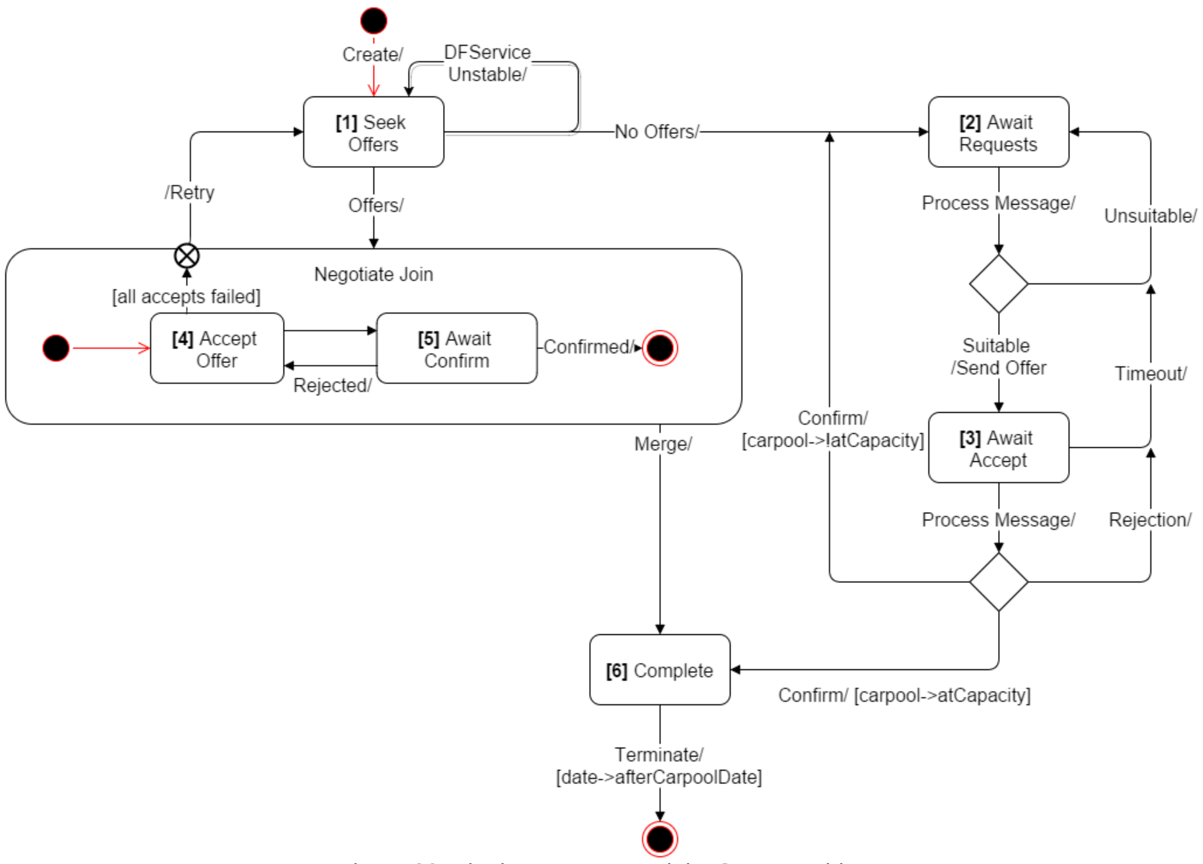


Figure 29: Final Route Agent Finite State Machine

The optionally injectable agents are:

1. The Evaluation Agent
   - Polls the database and provides metrics such as the cost of one user per carpool, actual cost of all carpools in the system, overall savings, or average savings per carpool.
   - Evaluations are executed as ticker behaviours, thus new metrics can easily be added and the polling frequency can be altered by modifying a single variable.
2. The DFService Monitoring Agent
   - Queries the DFService at regular intervals to determine the response time for the most complex possible search given the system's current state (specifically, all registered agents regardless of service name).
   - Updates the DFService Status singleton with the current status of the DFService Agent – one of: OK, slow or unstable.

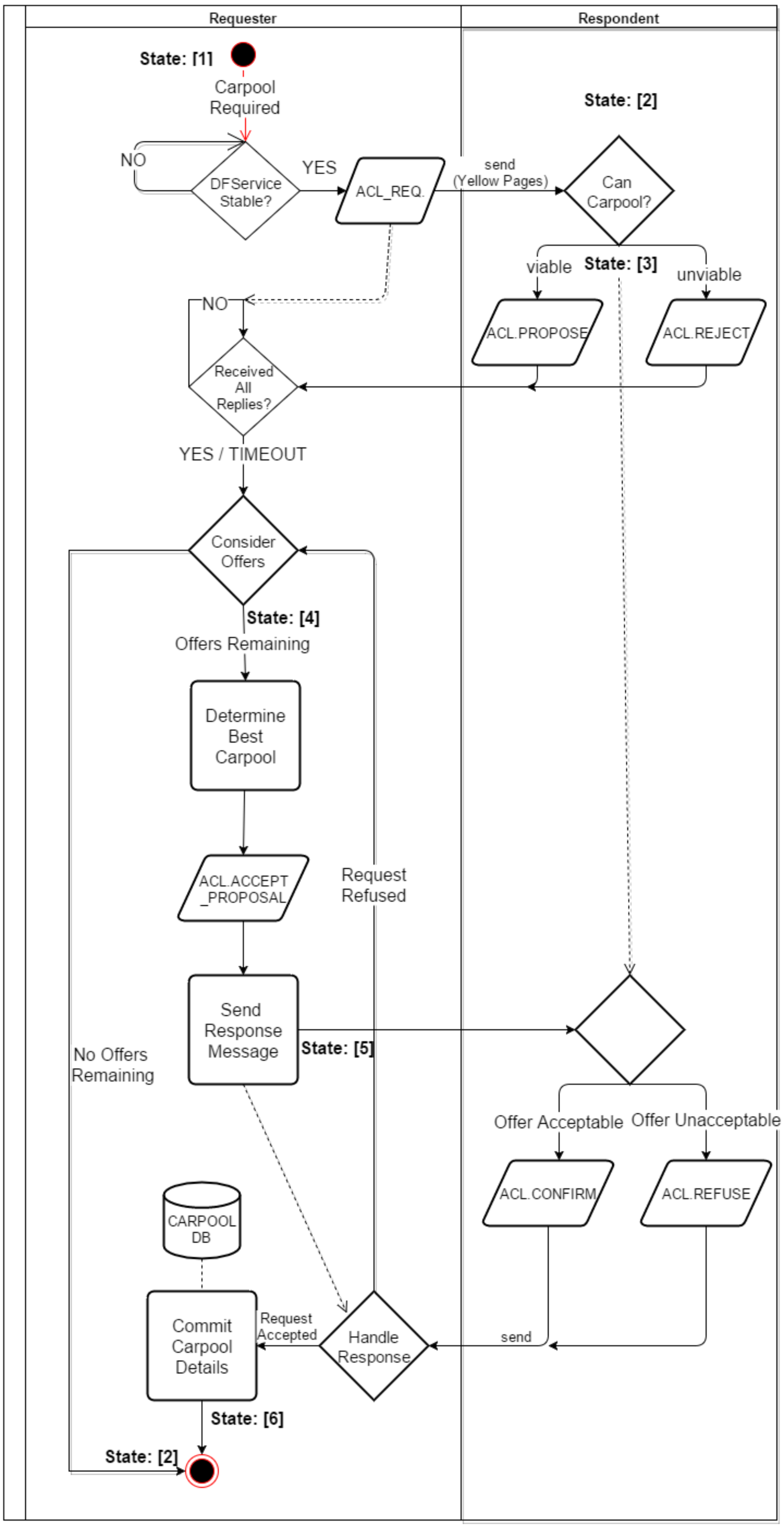


Figure 30: Final Carpool Negotiation Sequence Diagram

7.2. Database

The database remains unchanged from the initial design.

The data store is fundamentally composed of two tables: users and carpools. One limitation of PostgreSQL is its lack of support for arrays of foreign keys. The solution to this issue is to use an intermediary table which links user IDs to carpool IDs, allowing for example to get all members of carpool 100 with the query:

```
SELECT uid FROM ucintermediary WHERE cid='100';
```

A similar table is required to store carpool proposals – however it is important to note that the intermediary and proposal tables cannot be combined into one, as this requires an extra field to denote whether the entry is a proposal or an arranged carpool.

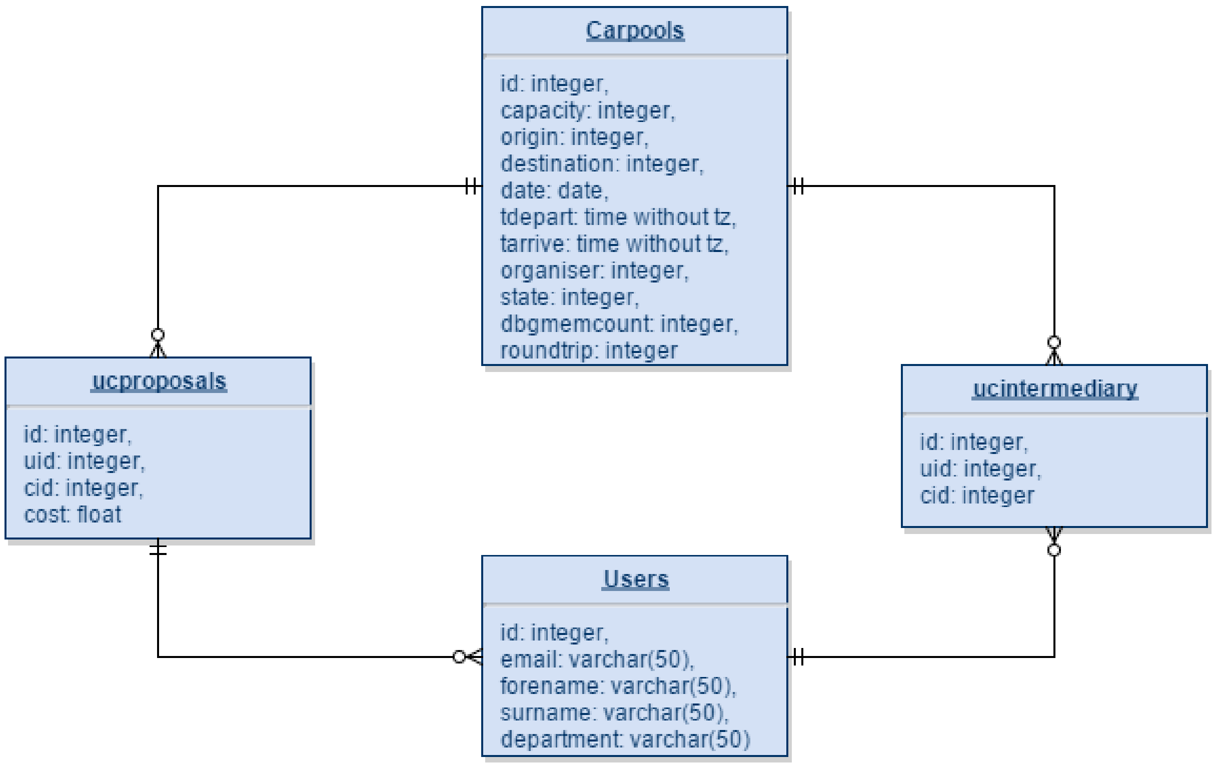


Figure 31: Final Entity-Relation Diagram

Two NOTIFY triggers are required by the agent system to prevent the need for periodic database polling – the benefits of this are that database listeners will be informed immediately (as opposed to whenever their next poll occurs) and the agents do not have to track their last read row in order to determine whether there has been an insertion since the last check. These triggers are on the Users and Carpools tables, calling a function which sends a NOTIFY to the table's channel containing the row ID to simplify the logic on the agent's end.

7.3. Application Programming Interface

The web API remains unchanged from the initial design.

The web API exists to provide a gateway to the backend which is immutable from the client application's perspective. It empowers backend developers to make drastic changes to business logic without requiring client updates, provided that the API endpoints remain the same and the wrapper scripts are updated to correctly bridge the two domains.

In order to meet the performance and availability requirements, some key decisions must factor into the design. The most significant of these is the architectural style of the web API. The system architecture proposed here will benefit from the use of a RESTful architectural style for a number of reasons:

1. REST APIs provide extremely good support for read caching, making the web service very efficient and minimising the data transfer required on clients which may be charged based on the amount of data transferred over a given period.
2. REST APIs lend themselves to scalable solutions due to the strict forbiddance of conversational state; clients do not care if the server they're talking to is different every time, therefore the web service does not have to track connections in any way. Overall performance is also improved due to the fact that all data required to respond to a request must be included in the request – there are no client-conditional lookups or calculations to be made.

To conform to the RESTful architectural style (Fielding, 2000), the web service must conform to the following criteria:

1. Interaction between client and server must be stateless between requests: there is no notion of ongoing conversation and all data required to execute a request must be provided in the request.
2. Clients can cache responses, therefore the web service should explicitly define responses as cacheable.
3. The system must be layerable, allowing for intermediary systems to be introduced to improve system scalability – for example a load balancer, central cache or web application firewall (WAF).
4. Provide a uniform interface which simplifies and decouples the architecture, through:
    a. Identification of resources in requests (general using URIs in a web API);

b. Ensuring manipulation of the above resources by clients is possible using only their stored representation of them – the client must be served all required information;
c. Self-descriptive messages which describe to the client how it should process the message, such as media type and cacheability; and
d. Explicitly defined state transition resources as part of a response, before which clients should not attempt state transitions to any resource which is not explicitly defined as a fixed entry point to the web API.

Method	URL	Data	Result
POST	/v1/register	Email Forename Surname User Tags	User registration.
GET	/v1/users		Returns a list of all system users.
GET	/v1/users/<int:user_id>		Returns the public details of a single user.
GET	/v1/carpools		Returns a list of all arranged carpools in the system.
GET	/v1/carpools/<int:user_id>		Returns a list of carpools involving a single user.
GET	/v1/carpools/<int:carpool_id>		Returns details of a single carpool.
POST	/v1/carpools	Capacity Origin Destination Date Departure Time Arrival Time Round Trip	Inserts the details for a new carpool search, settings the organiser as the authenticated user.
GET	/v1/intermediaries		Returns a list of intermediary table entries between users and carpools.
POST	/v1/proposals	UID CID	Marks a proposal as accepted by the authenticated user.

Figure 32: Final API Endpoint Specification

The API entry point design stems from the use cases of the client (which in turn stem from the use cases of the end user). As specified in the requirements analysis, these are:

1. Authenticate
2. View carpool details
3. Initiate carpool searches
4. Accept carpool offers

Which, with regards to the web API, imply a number of subgoals required for clients to provide system users with the required information to interact with the system:

1. Retrieve details of the end user
2. Retrieve a list of intermediary table entries (as specified above, these are used in the database to emulate an array of foreign keys)
3. Retrieve the details of carpools which the end user is a member of
4. Retrieve all proposals which require the end user's attention

It should be kept in mind that clients may not use all of these endpoints, and indeed a number of endpoints should not be exposed to the end user applications – these are easily identified as the calls which do not return conditional information based on the provided user ID. These endpoints have been included in the specification as they are extremely useful for administration, maintenance and debugging.

As the API will be exposed over the internet to support roaming client applications, security is a critical consideration. To ensure user data is accessible only by the user it relates to, an open standard for client authorisation should be implemented – such as OAuth 1.0 or Security Assertion Markup Language (SAML) 2.0. Importantly, the implementation should enforce a secure communication protocol such as Hypertext Transfer Protocol (HTTP) over Secure Sockets Layer (SSL), commonly referred to as HTTPS. With this enforcement should come secure development decisions to protect the end user such as certificate pinning or end-to-end encryption which prevent man-in-the-middle attacks.

However, securing the client does not secure the API itself. Standards to verify the integrity of a message (that is, to verify that it has not been tampered in transit by an attacker or spoofed using command line tools such as cURL) are well established: key-hash message authentication codes (HMAC) allow messages to be signed using a secret key which is known only to the client application, preventing arbitrary requests from untrusted sources while also verifying message integrity in transit. The server holds the same private key and can use it to recalculate the message hash for verification purposes – and to sign a response which the client can then verify in the same manner. This provides two-way trust and ensures that the server itself is not an easy attack vector.

7.4. Deployment

The deployment differs from the initial design in three key ways:

1. *The DF Agent and AMS Agent are isolated in the Main-Container. Agents spawned by the carpooling system should never spawn in Main-Container.*
2. *There exists a Monitoring container reserved for optionally injectable agents and the Agent Factory which monitors the agent system's external environment (the database).*
3. *Each container should represent a single date, preferably with its host system scaling vertically to prevent the communication overheads discussed by Cortese et al. (2002).*

The scalable and decoupled nature of the architecture, combined with its tendency towards web-facing implementations, lends itself to a cloud computing deployment. Not only does this provide a convenient and hardware-free deployment – but also the ability to deploy globally (perhaps for corner cases such as group airfares), ensure component redundancy, scale automatically in real-time (no lead time on hardware purchasing), and remove large maintenance workloads from the internal teams.

The architecture proposed in this paper will be based upon Amazon Web Services for reasons detailed in Section 2.4. The architecture is deployable on most similar cloud computing providers – such as Google, Microsoft and IBM – as well as on any heterogeneous network whether it is in-house, cloud-based or hybrid.

The terminology used in Amazon Web Services deployment can be confusing at times. For the benefit of understandable deployment on arbitrary infrastructure, a brief explanation will be given here:

- EC2 – Elastic Compute Cloud: Resizable cloud computing instances (virtual machines) which support the booting of machine images to automate the configuration and deployment of applications.
- Route 53: High availability scalable Domain Name System (DNS) which supports modification of records through web service calls, such as adding a new EC2 instance to a hosted zone programmatically when it comes online.
- Elastic IP: Static IP addresses designed to improve availability through programmatic remapping of public IP addresses to instances (for example to mask failures without having to wait for DNS to propagate to clients).

- S3 – Simple Storage Service: Secure, durable and highly-scalable object storage. File containers in S3 are referred to as buckets, with each bucket having a globally unique identifier and being globally distributable.
- ELB – Elastic Load Balancing: Automatic distribution of incoming traffic across EC2 instances, providing high levels of fault tolerance and dynamically scaling load balancing capacity based on traffic levels. ELB also supports auto scaling of instances based on user-defined conditions, using metrics such as average CPU load and minimum latency.
- Availability Zone: An isolated datacentre in a given region – allowing developers to deploy in multiple availability zones per region for higher availability in case of outages.
- Security Group: Virtual firewalls which control traffic for their associated instances.

The ideal deployment of the architecture is in a virtual private cloud, functioning as a private network with none of the backend exposed whatsoever – there is only one path into the network which is the web API itself. From here the traffic may be verified with the integrity checks suggested in Section 4.3, restricting access to the network to trusted clients only. The API will also handle authorisation checks at this stage to ensure that the user is entitled to access the backend resources which they are requesting, executing the relevant query if successful.

The database itself is mirrored in a separate availability zone to allow quick migration of the system in case of an outage, with periodic database dumps being stored in an S3 bucket to enable disaster recovery (such as database corruptions which are mirrored across availability zones). Database updates are broadcast to all JADE EC2 instances, while new carpool requests will be sent via the load balancer to ensure that the new agent spawns on the least taxed system.

CloudWatch allows the auto scaling cluster of containers to expand and contract as necessary, providing alarms which can be sent to agents on instances which are about to be destroyed or to system administrators when a problem is detected.

In the ideal scenario, each container has its own EC2 instance which scales elastically to meet demand. The DF Agent and AMS Agent are the only agents which should exist in the Main-Container, with any agent spawned by the system living in a non-default container. Critically, the DFService Monitoring agent should live in a container which is as remote as the most remote Route Agent container, otherwise its checks will be inaccurate.

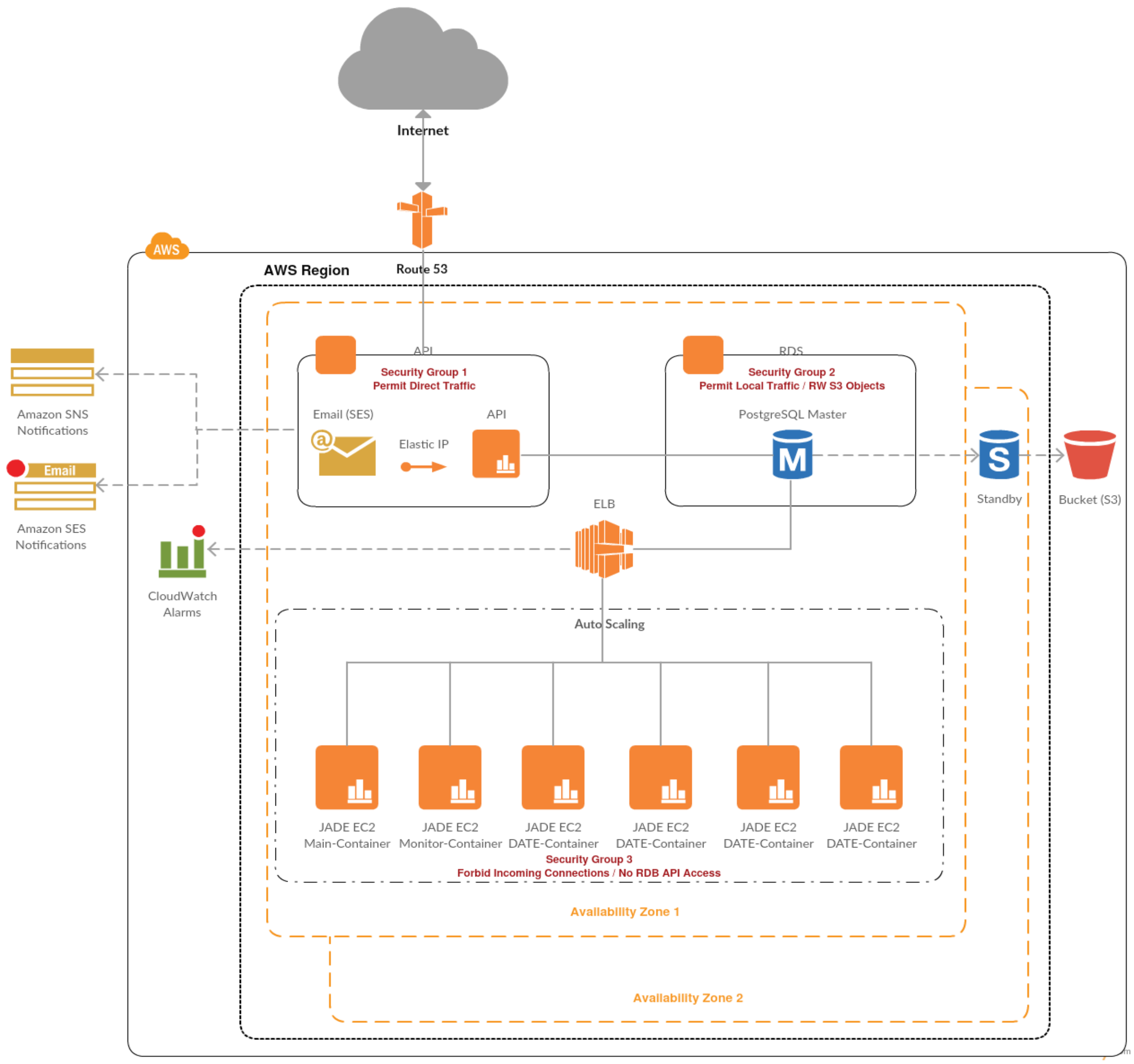


Figure 33: Final Architecture Deployment

7.5. Client Application

The client application remains unchanged from the initial design.

Client applications within the system can take almost any form, but in order to best meet the flexibility and usability requirements there are some clear decisions to be made. The two clear types of client application are websites and mobile applications. The design decisions for these are significantly different as the use cases are polar opposites: websites are likely to be accessed from a desktop or laptop on the organisation's private network, implying that the entire system can be hosted on an intranet to remove the majority of security concerns – however this use case does not make much sense as the nature of carpools mean that the system user will likely not have a workstation at one end of the journey. Mobile applications on the other hand allow the user to take the details of their carpools with them, providing a convenient method of double checking the arrangements when with an intranet site they could not. Integrations such as phone numbers (for use cases such as making contact with the carpool members and arranging specifics such as where to meet, or chasing up members who may be running late) make far more sense on mobile platforms, and powerful server-side options such as push notifications become available to make the system far more convenient for end users. For these reasons the architecture design will focus on mobile applications for end users – but this does not mean other client types are not possible. As long as a platform's stack supports, or can be modified to support, RESTful API consumption, the client can be developed to target that platform.

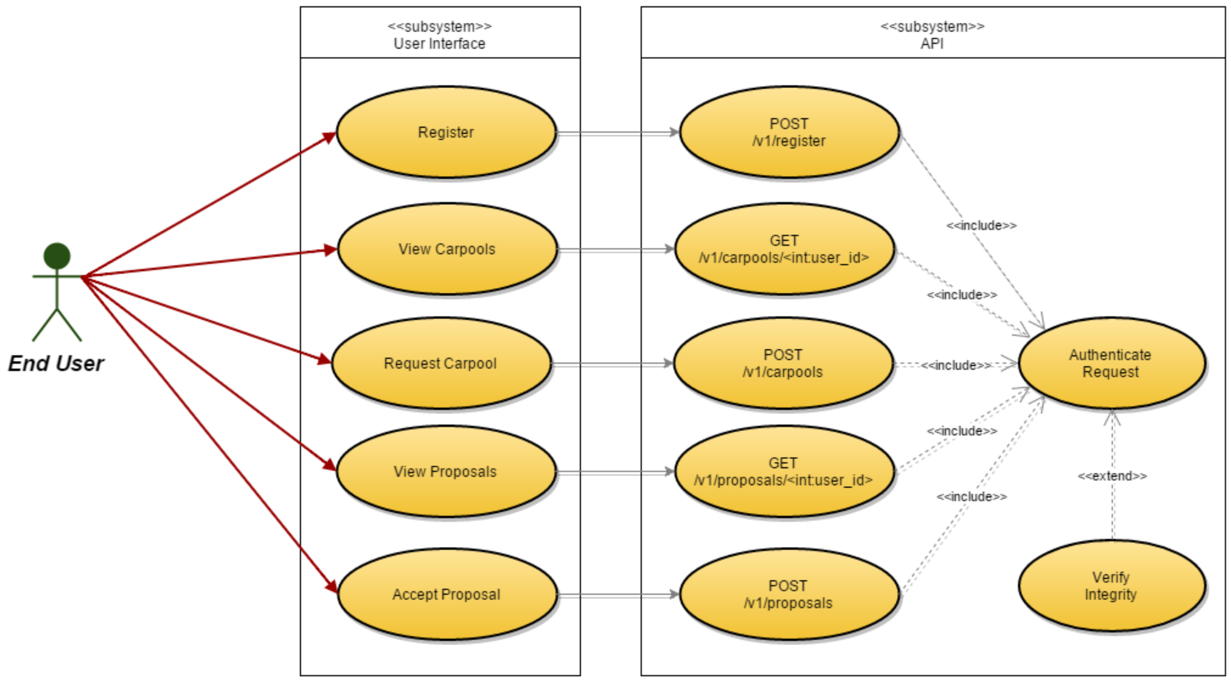


Figure 34: Final End User Application Use Case

Mobile applications which are for internal organisational use are unlikely to be distributed through application stores therefore the design must provide a method of easily updating all clients without manual intervention on every installed device. This implies the design should favour either HTML5 mobile applications or hybrid mobile applications – both of which are approaches supporting deployment of one implementation across multiple platforms (Android, iOS, Windows Phone, FirefoxOS). However given the usability requirements, hybrid applications clearly stand out as the best choice for the system architecture. Hybrid applications not only allow the application to be updated on the fly by implementing a full web application on the server with the application simply acting as a wrapper for this deployment, but also provide access to native capabilities such as standard system gestures, GPS and much more. Frameworks such as Ionic and Apache Cordova make it extremely easy to develop hybrid applications which retain a strong native focus across supported platforms, allowing convenient access to the native device APIs when required. Both of the example frameworks also support live reloading of deployed apps, making it extremely simple to update the entire installed device base without requiring any user interaction – as well as pseudo app stores to allow private deployment of hybrid applications.

8. Conclusion

It is important to critically evaluate the architecture specifically with regards to the agent technology used. Heterogeneous architectures, cloud architectures and agent architectures are all well established as being viable architectures for a wide range of purposes. Specifically, then, the following points must be concluded upon: how relevant and useful agent technology is in a *carpooling* architecture; how successful the proposed architecture is at meeting the requirements specified in Section 4, based on previous research reviewed in Section 2; and how the proposed architecture compares to previous attempts.

Agent technology and the JADE framework have proven to bring a number of benefits to a carpooling architecture:

- The ability to develop and deploy an asynchronous system across heterogeneous servers with relative ease – which is inherently maintainable, portable and scalable;
- The distribution of carpool negotiation is perhaps the clearest benefit of such an architecture: instead of a single component considering tens of thousands of potential matches, it instead becomes tens of thousands of components considering one single request. This provides the user with an extremely responsive system regardless of the size of their search, assuming the architecture has been scaled correctly to manage the load;
- Autonomous behaviour on behalf of a user is very appealing in a system which is always-on, and is extremely difficult without agents which can reason about the reputation of other agents involved. If a user simply wants to establish a carpool based on their desired criteria – that is, they do not wish to review the offers – then the system can work entirely autonomously for them and reliably negotiate the best valid carpool not only in terms of price but also in terms of reputation;
- Fluctuations in network connectivity between containers or agents is no longer an issue. A container or agent can go completely offline and, while it will be unable to negotiate with other agents, it will not lose information or cause system instability for any users other than its owner; and
- The ability to spawn and destroy agents which fulfil specific requirements in the system, without causing the system to restart in either case. A good example of this is the DFService Monitoring agent detailed above, which is entirely optional within the system but brings with it certain benefits such as managing DFService load in deployments which attempt to maximise agents per container.

However there are also some clear disadvantages to agent technology in a carpooling system:

- The behaviour of the system is extremely unpredictable as agents will send and respond to messages in a stochastic order with each execution, purely due to the fact that each agent runs on a dedicated thread and the processor scheduling cannot be predicted. This makes it difficult to test the system in a consistent manner;
- Correctly load balancing an agent system is an extremely complicated task. Simply selecting the container with the least agents does not work as this container may be the host of all of the most active agents, while selecting the least busy container may be the correct short-term decision but the incorrect long-term decision due to the dynamic nature of the agent system; and
- Scaling down is a development project of its own. Destroying a container will destroy its agents, therefore agents must be migrated before destroying a container. It is extremely difficult to predict whether this migration will push another container over the acceptable load threshold as it can be difficult to programmatically determine the load contribution of each agent – as this involves predicting the actions of their users.

In conclusion, then, the advantages of agent technologies – specifically the JADE platform – with regards to carpooling system architectures appear to greatly outweigh the drawbacks. The architecture meets all specified requirements, appears to function more than acceptably under unrealistic stress on the least powerful cloud computing instances, and inherently supports a number of concepts associated with carpooling such as negotiation, reasoning and lifecycles. With strong future development projects it seems feasible that at least the last two drawbacks can be resolved and even turned into advantages.

8.1. Future Work

As the scope of this project was so constrained, the proposed architecture was evaluated at a very high level using a reduced functionality prototype. There are a large number of aspects of the architecture and hooks in the prototype which could be built upon for further experimentation, validation and evaluation. In order of perceived importance, these are:

- The viability of the architecture in a scenario which allows carpools from arbitrary start points to arbitrary end points, requiring the use of a geographic coordinate system and map technologies; and
- The effects of predictive load balancing on the performance of the system as a whole;
- The benefits of implementing a DF Agent hierarchy rather than a single agent handling all search requests;
- Validation of the theories and design regarding automatic scaling of the architecture in a cloud deployment – and whether the system can truly scale infinitely without manual intervention;
- How complex reputation models such as REGRET can improve the quality of matches, as well as the advantages and disadvantages this has with regards to software quality;
- More complex carpool matching algorithms with additional features such as routes within routes, or multi-stage carpools in which a user may transfer from one carpool to another at some point in the journey.

8.2. Personal Evaluation

This project has been a great success in my eyes – a lot has been learned (at least personally) about technologies, deployment approaches and the major factors which prevent widespread adoption of carpooling schemes. All of these were bundled into a novel architecture which has, with some struggle, proven itself to be extremely scalable – both horizontally and vertically – as well as meeting all identified requirements.

Unfortunately, the scope of the project was perhaps the least manageable part of the process and continued to be an erratic factor throughout. I set out with ambitions to analyse every aspect of agents in a carpooling system and quickly realised that the field is far more complex than it first appears. The goal was to have strong load balancing, extremely complex route matching algorithms, a correct and complete implementation of a relevant agent reputation model, and so much more – each of which could arguably constitute an honours project on their own. For the most part this can be attributed to the fact that the JADE platform is extremely complex and interesting – with unexpected benefits and drawbacks to be discovered at every corner. In particular the directory facilitator (DF) issues detailed by Mengistu et al. (2008) became a focal point of evaluation and research, as their solution of bypassing the DF for local-container searches seems to cripple the distributable nature of the architecture which is arguably its biggest advantage over a non-agent system.

However after much descoping the end product of this project was a complete architecture which has been tested, validated and evaluated very widely but not particularly deeply. The reputation is basic, start and endpoints are restricted, and many of the benefits of cloud computing were left untouched – but in the grand scheme of things the question of the viability of agents in a carpooling system (specifically *not* slugging, flexible, or dynamic carpooling, but rather the traditional prearranged approach to ride sharing) was answered. It is possible, it is viable, and all signs point to it in fact being superior in many aspects such as scalability, maintainability, reusability, portability and performance. However the drawbacks of using a generic agent framework made an appearance time and time again, perhaps going to show that this project has only seen the tip of the iceberg.

To put the personal success into perspective, it ironically feels best to summarise every aspect of the project which was not a deliverable (but rather a personal achievement or area of growth). I have learned about autonomous and intelligent agents from a background of zero knowledge; I have implemented a Python Flask API which supports all of the features exhibited by current bleeding edge developments such as Facebook's Graph API and the Twitter

Developer API; I have learned about hybrid application development, implementing a fully featured Ionic application purely as a tangible demonstration of the architecture – which can be deployed on a wealth of platforms despite being written only once; I have learned the intricacies of Amazon Web Services such as cloud monitoring and programmatic service deployment; and perhaps most importantly I have learned to manage a project with zero external management beyond suggestions and advice provided in the weekly progress meetings.

It is hard not to be satisfied with such personal improvement and I hope this work will contribute significantly to the fields of both software engineering and sustainable development, whether that comes in the form of subsequent research or simply inspiring other undergraduates who feel at peace only when faced by overwhelming project scope.

References

Abrahamse, W., & Keall, M. (2011). *Weaving a local web: Evaluating the effectiveness of Let's Carpool to encourage carpooling to work*. New Zealand Centre for Sustainable Cities: Otago University, Wellington.

Abrahamse, W., & Keall, M. (2012). Effectiveness of a web-based intervention to encourage carpooling to work: A case study of Wellington, New Zealand. *Transport Policy*, *21*, 45–51.

Amazon Web Services (AWS). (2015). AWS named as a leader in the IaaS Magic Quadrant for the 5th consecutive year. Retrieved November 23, 2015, from https://aws.amazon.com/resources/gartner-2015-mq-learn-more/

Artiach, T., Lee, D., Nelson, D., & Walker, J. (2010). The determinants of corporate sustainability performance. *Accounting & Finance*, *50*(1), 31–51.

Avram, M.-G. (2014). Advantages and challenges of adopting cloud computing from an enterprise perspective. *Procedia Technology*, *12*, 529–534.

Baldassare, M., Ryan, S., & Katz, C. (1998). Suburban attitudes toward policies aimed at reducing solo driving. *Transportation*, *25*(1), 99–117.

Bellifemine, F., Poggi, A., & Rimassa, G. (2001). Developing multi-agent systems with a FIPA-compliant agent framework. *Software-Practice and Experience*, *31*(2), 103–128.

Bento, A. M., Hughes, J. E., & Kaffine, D. (2013). Carpooling and driver responses to fuel price changes: Evidence from traffic flows in Los Angeles. *Journal of Urban Economics*, *77*, 41–56.

BlaBlaCar. (2015). BlaBlaCar - Trusted ridesharing. Retrieved November 4, 2015 from https://www.blablacar.co.uk/

Bolton, P. (2014). *Petrol and diesel prices*. House of Commons standard note SN/SG/4712, London: House of Commons Library.

Bond, A. H., & Gasser, L. G. (Eds.). (1988). *Readings in distributed artificial intelligence*. San Mateo, CA: M. Kaufmann.

Chmiel, K., Tomiak, D., Gawinecki, M., Karczmarek, P., Szymczak, M., & Paprzycki, M. (2004). Testing the efficiency of jade agent platform. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on* (pp. 49–56). IEEE.

Cho, S., Yasar, A.-U.-H., Knapen, L., Bellemans, T., Janssens, D., & Wets, G. (2012). A Conceptual Design of an Agent-based Interaction Model for the Carpooling Application. *Procedia Computer Science*, *10*, 801–807.

Cortese, E., Quarta, F., Vitaglione, G., & Vrba, P. (2002). Scalability and performance of jade message transport system. In *AAMAS Workshop on AgentCities, Bologna* (Vol. 16).

Department of Energy and Climate Change. (2014). *Annual energy statement 2014*. Retrieved February 12, 2015 from http://www.gov.uk/government/publications

Drifty Co. (2015). Ionic: Advanced HTML5 Hybrid Mobile App Framework. Retrieved November 23, 2015, from http://ionicframework.com/

Egenhofer, M. J. (1993). What's special about spatial?: database requirements for vehicle navigation in geographic space. *ACM SIGMOD Record*, *22*, 398–402.

El Fallah Seghrouchni, A., Dix, J., Dastani, M., & Bordini, R. H. (Eds.). (2009). *Multi-Agent Programming:* Boston, MA: Springer US.

Federal Highway Administration. (2010). *Casual Carpooling Scan Report (Exploratory Advanced Research Program).* Washington, DC: U.S. Department of Transportation.

Google. (2015). Google Maps API licensing - Google Maps API — Google Developers. Retrieved February 28, 2015, from https://developers.google.com/maps/licensing

GraphHopper. (2015). GraphHopper Route Planner. Retrieved March 1, 2015, from https://graphhopper.com/

Hall, B., & Leahy, M. G. (2008). Open Source Approaches in Spatial Data Handling - Brent Hall, Michael G. Leahy - Google Books. In *Open Source Approaches in Spatial Data Handling* (pp. 105–129).

Hess, J. M. (2011). *Young Future Mobility Leaders - Panel*. Presented at the Ecosummit 11, Berlin, Germany.

Hosam, A.-S., Abbas, M., Ahmad, S., & Mustafa, A. (2010). Intelligent Agent System Architecture for Presenting Health Grid Contents from Complex Database (pp. 38–42). Presented at the International Conference on Intelligent Systems, Modelling and Simulation (ISMS), Liverpool, England: IEEE.

ITO World. (2015). OSM Analysis Summary. Retrieved November 23, 2015, from http://itoworld.com/product/data/osm_analysis/main

Jennings, N., & Wooldridge, M. J. (Eds.). (1998). *Agent technology: foundations, applications, and markets*. Berlin ; New York: Springer.

Khosravifar, B., Bentahar, J., Gomrokchi, M., & Alam, R. (2012). CRM: An efficient trust and reputation model for agent computing. *Knowledge-Based Systems*, *30*, 1–16.

Koppelman, F. S., Bhat, C. R., & Schofer, J. L. (1993). Market research evaluation of actions to reduce suburban traffic congestion: Commuter travel behavior and response to demand reduction actions. *Transportation Research Part A: Policy and Practice*, *27*(5), 383–393.

Kothari, A. B. (2004). Genghis-a multiagent carpooling system. *The University of Bath, B. Sc. Dissertation Work in Computer Science, UK*.

Kray, C. (2001). The Benefits of Multi-Agent Systems in Spatial Reasoning. In *FLAIRS Conference* (pp. 552–556).

Lo Piccolo, F., Bianchi, G., & Salsano, S. (2006). Measurement study of the mobile agent jade platform. In *Proceedings of the 2006 International Symposium on World of Wireless, Mobile and Multimedia Networks* (pp. 638–646). IEEE Computer Society.

Mell, P., & Grance, T. (2011).*The NIST definition of cloud computing*. Gaithersburg, Maryland: National Institute of Standards and Technology (NIST).

Mengistu, D., Tröger, P., Lundberg, L., & Davidsson, P. (2008). Scalability in Distributed Multi-Agent Based Simulations: The JADE Case (pp. 93–99).

OpenLayers. (2015). OpenLayers 3 - Welcome. Retrieved March 1, 2015, from http://openlayers.org/

OpenStreetMap. (2015). OpenStreetMap. Retrieved March 1, 2015, from http://www.openstreetmap.org/copyright

Oxford University Press. (2015a). carpool - definition of carpool in English from the Oxford dictionary. Retrieved November 5, 2015, from https://www.oxforddictionaries.com/definition/english/carpool

Oxford University Press. (2015b). ride-sharing - definition of ride-sharing in English from the Oxford dictionary. Retrieved November 5, 2015, from https://www.oxforddictionaries.com/definition/english/ride-sharing

pgRouting Community. (2015). pgRouting Project. Retrieved November 11, 2015, from http://pgrouting.org/

Pisarski, A. E. (2006).*Commuting in America III: The Third National Report on Commuting Patterns and Trends*. Washington: Transportation Research Board.

Ray, S., Simion, B., & Brown, A. D. (2011). Jackpine: A benchmark to evaluate spatial database performance (pp. 1139–1150). Presented at the IEEE 27th International Conference on Data Engineering (ICDE), IEEE.

Sabater, J., Paolucci, M., & Conte, R. (2006). Repage: REPutation and ImAGE Among Limited Autonomous Partners. *Journal of Artificial Societies and Social Simulation*, *9*(2), 3.

Sabater, J., & Sierra, C. (2002). Reputation and social network analysis in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1* (pp. 475–482). ACM.

SAP SE. (2015). TwoGo | Ridesharing & carpooling for your daily commute. Retrieved November 4, 2015, from https://www.twogo.com/

Sghaier, M., Zgaya, H., Hammadi, S., & Tahon, C. (2010). A distributed dijkstra's algorithm for the implementation of a real time carpooling service with an optimized aspect on siblings. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on* (pp. 795–800).

sysware. (2015). Carpooling Software | Comovee Mobility Management. Retrieved November 4, 2015, from http://www.comovee.com/

TaxiFareFinder. (2015). TaxiFareFinder - Estimate You Taxi Cab Fare, Cost & Rates. Retrieved November 23, 2015, from http://www.taxifarefinder.com/

TIME. (2015). Oxford Dictionaries Adds Janky, EGOT and Ridesharing. Retrieved November 5, 2015, from http://time.com/3724601/oxford-dictionary-janky-egot-ridesharing/

Wosskow, D. (2014).*Unlocking the sharing economy: An independent review*. Department for Business, Innovation and Skills.