## Technology Comparisons

Having corresponded via e-mail with Jim Brikman, previously a senior developer at LinkedIn, in relation to comparing Node.js with other frameworks, he suggested that technologies can’t be compared on any one single point but must be measured against each other according to a number of different criteria and offered the following list as a suggestion (Brikman, 2015):

1. **Learn**: getting started, ramp up, overall learning curve.
2. **Develop**: routing, templates, i18n, forms, json, xml, data store access, real time web.
3. **Test**: unit tests, functional tests, integration tests, test coverage.
4. **Secure**: CSRF, XSS, code injection, headers, authentication, security advisories.
5. **Build**: compile, run tests, preprocess static content (sass/less/CoffeScript), package.
6. **Deploy**: hosting, monitoring, configuration.
7. **Debug**: step by step debugger, profilers, logging,
8. **Scale**: throughput, latency, concurrency.
9. **Maintain**: code reuse, stability, maturity, type safety, IDEs.
10. **Share**: open source activity, mailing lists, popularity, plugins, commercial support, jobs.

Whilst this is undoubtedly a very comprehensive set of comparison parameters, given the timescale available it will be necessary to limit the scope of this project to no more than two of the above headings (with some minor modifications), plus one more. These will be as follows:

1. **Learn**: Getting started, available resources, tutorials etc.
2. **Footprint**: Disc space occupied by base code plus dependencies, CPU consumption.
3. **Scale**: Concurrency, throughput, latency.

### Learn

Node.js, Ruby, Ruby on Rails, and PHP all have excellent resources available, both online, in the form of video tutorials and in soft and hard copy. A quick search on YouTube using just “Node.js Tutorials”, “PHP Tutorials” and “Ruby Tutorials” returned 174,000 results for Node, 118,000 results for Ruby and 990,000 results for PHP (YouTube.com, 2015). Whilst these results are far from scientific and assuming that there’s a full spectrum of content ranging in quality from excellent to extremely poor, one could possibly conceivably progress from a complete novice to gain a reasonable understanding of any of the above technologies from YouTube alone. Separate and apart from self-directed learning opportunities, Node.js, PHP and Ruby have vibrant online communities consisting of user groups who communicate via blogs and meetups, where they share knowledge, with, as well as learn from, each other.

The lack of a good IDE for Node.js, as expressed by Kiran Prasad of LinkedIn (Norton & Coatta, 2014), is something which sets Node at a disadvantage when compared to both Ruby on Rails and PHP. Although there are several cloud based IDE’s such as Cloud9 and Koding, as well as native text editors such as SublimeText and WebStorm. However, whilst these offer varying degrees of excellence in terms of text editing and syntax colour coding, there’s still a conspicuous absence of comprehensive debugging tools to enable a developer to trace bugs to a definable source, as can be found in NetBeans for Java development and Visual Studio for .NET and C#. Yet despite this shortcoming, Prasad still favours Node.js over Ruby on Rails, based on its greater performance, smaller [memory] footprint and the significantly fewer lines of code required to achieve a desired result (Norton & Coatta, 2014).

### Footprint

To be established when testing and comparing applications.

### Scale

#### Concurrency

Prior to the arrival of Node.js in 2009, Paulson, in 2007, identified dynamic programming languages as growing in popularity, she also quotes Tim Bray, the then director of Web Technologies at Sun, as noting that the requirements of large software projects are beyond the capabilities of many dynamic languages but that “*some of their agility-related features are slowly percolating into enterprise languages such as Java*” (Paulson, 2007).

### Threading/Threads

Ruby has had the capacity to generate multithreaded code from the beginning and, as an indication of how well thought out the language is, will support multithreading across all platforms irrespective of whether the OS upon which it’s operating supports this (ruby-lang.org, 2015). Despite the fact that Purer (Purer, 2009) in 2009 states that PHP doesn’t support threads, PHP’s own documentation would suggest this to be incorrect with references to threads and multithreads appearing in their changelog dating back to Version 4.0 Beta 2 in Aug. ’99 (PHP, 2008). Node.js on the other hand eschews multithreading and runs on a single thread operating a continuous event loop, this effectively dispenses with the need for concurrent threads, thereby freeing up resources, enabling I/O intensive applications to process requests in near real-time (Node.js, 2015). Although this approach offers unmatched performance, it also represents a single point of failure, with one blocking function, in the absence of threading, having the potential to bring the whole operation to a halt until this particular process has completed (Brikman, 2014).

### Non-Blocking Event-Driven I/0

Node is the only technology, of the three being compared, which is built solely around the concept of event-driven non-blocking I/O (Dahl, 2011). The Ruby community offers a number of frameworks that can deal with concurrency using an event loop to give asynchronous non-blocking performance, EventMachine and Celluloid::IO being two examples (GitHub.com, 2011) whilst PHP enables its developers to produce event-driven non-blocking I/O functionality via ReactPHP (GitHub.com, 2012). Thruway is another PHP based non-blocking technology however this framework uses ReactPHP at its core and as such is simply an extension of ReactPHP (Dan, 2015). Both EventMachine and ReactPHP make use of the Reactor Design Pattern to achieve non-blocking, asynchronous I/O performance equivalent to that of Node.js. However whilst a certain level of non-blocking asynchronous I/O is achieved using this pattern, there are limits to its capabilities, namely a default limit on concurrent requests of 20 (Igvita, 2008), also the Reactor Pattern effectively gives an illusion of asynchronous behaviour whilst in the background delivering requests concurrently to a service handler, demultiplexing them and distributing them synchronously to their various relevant handlers.