Technology Radar

http://www.thoughtworks.com/radar

Prepared by the ThoughtWorks
Technology Advisory Board

MARCH 2012

ThoughtWorks®

What's new?

Since the last publication of the Technology Radar, these technology trends are most prominent:

- Continued development of alternatives to SQL datastores
- Treating all code from UI to tests with respect
- Increasing diversification and rigor in browser based languages and technologies
- Smaller, simpler and faster applications and services

Introduction

The ThoughtWorks Technology Advisory Board is a group of senior technology leaders within ThoughtWorks. They produce the ThoughtWorks Technology Radar to help decision makers understand emerging technologies and trends that affect the market today. This group meets regularly to discuss the global technology strategy for ThoughtWorks and the technology trends that significantly impact our industry.

The Technology Radar captures the output of these discussions in a format that provides value to a wide range of stakeholders, from CIOs to enterprise developers. The content provided in this document is kept at a summary level, leaving it up to the reader to pursue more detail when needed. The goal of the radar is conciseness, so that its target audience understands it quickly. The radar is graphical in nature, grouping items into techniques, tools, languages and platforms. Some radar items could appear in multiple quadrants, but we mapped them to the quadrant that seemed most appropriate. We further group these items in four rings to reflect our current position on them.

The rings are:

- **Adopt:** We feel strongly that the industry should be adopting these items. We use them when appropriate on our projects.
- **Trial:** Worth pursuing. It is important to understand how to build up this capability. Enterprises should try this technology on a project that can handle the risk.
- **Assess:** Worth exploring with the goal of understanding how it will affect your enterprise.
- **Hold:** Proceed with caution.

Items that are new or have moved since the last radar are represented as triangles, while items that have not moved since the last radar are represented as circles. As we look at each quadrant in detail, we show the movement that each item has taken since the last publication of the radar. Items that have not moved recently fade and are no longer displayed unless something significant happens.

Contributors

The ThoughtWorks Technology Advisory Board is comprised of

Rebecca Parsons (CTO)
Martin Fowler (Chief Scientist)
Nick Hines (CTO Innovation)
Evan Bottcher
Graham Brooks
Ian Cartwright
Erik Doernenburg
Ronaldo Ferraz
Jim Fischer
Neal Ford
Ajey Gore
Wendy Istvanick
Badri Janakiraman
James Lewis
Mike Mason
Sam Newman
Pramod Sadalage
Scott Shaw
Hao Xu
Jeff Norris

Techniques

Adopt
- 1. Automate database deployment
- 2. Coding architects
- 3. Continuous Delivery (CD)
- 4. Data visualizations of development and operations
- 5. DevOps
- 6. Emergent design
- 7. Evolutionary architecture
- 8. Evolutionary database
- 9. Health check pages
- 10. Simple performance trending
- 11. Test at the appropriate level

Trial
- 12. Acceptance test of journeys
- 13. Agile analytics
- 14. Build your own radar
- 15. Categorization & prioritization of technical debt
- 16. Embedding a servlet container
- 17. Event APIs
- 18. Infrastructure automation of development workstations
- 19. Out-of-container functional testing
- 20. Performance testing as a first-class citizen
- 21. Single command deploy
- 22. Thoughtful caching
- 23. Windows infrastructure automation

Assess
- 24. Event sourcing
- 25. Experience Design (XD)
- 26. Mechanical sympathy
- 27. Micro-services
- 28. Production immune system

Hold
- 29. Database based integration
- 30. Feature branching
- 31. Manual infrastructure management
- 32. Scrum certification
- 33. Server / application container end-of-life
- 34. Test recorders

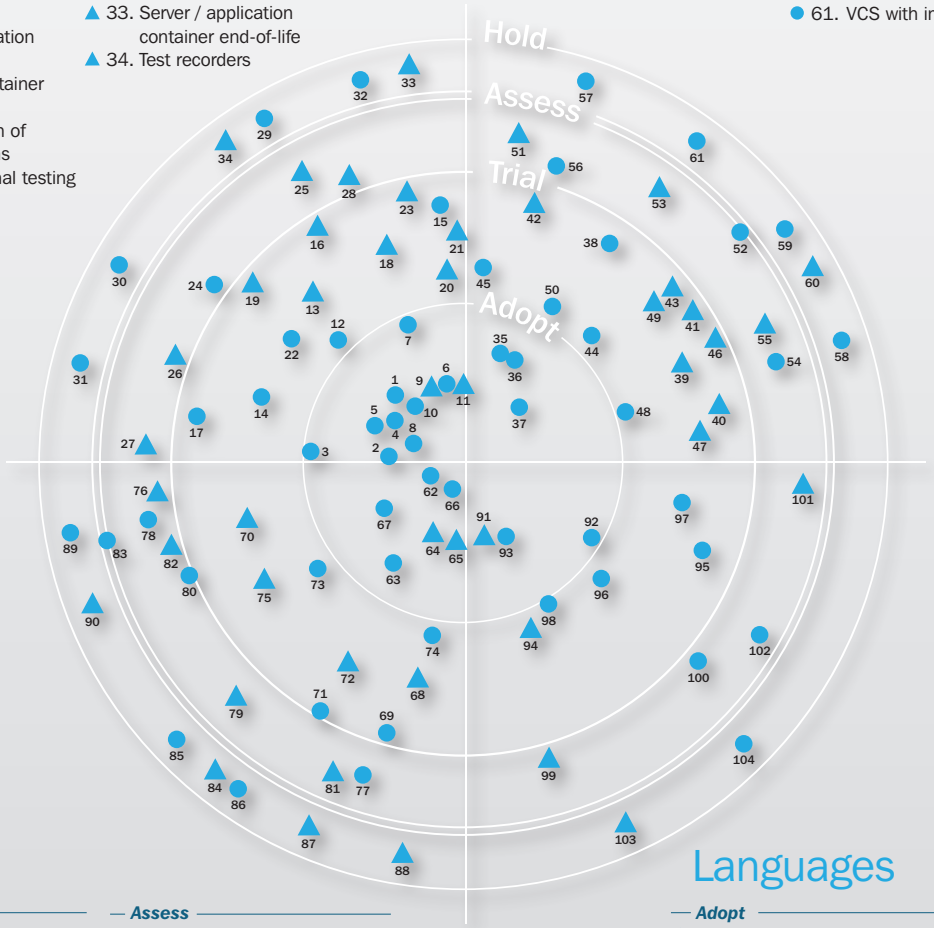
Tools

Adopt
- 35. Git
- 36. Github
- 37. Infrastructure as code

Trial
- 38. Client-side MVC
- 39. FPM
- 40. Frank
- 41. Jade
- 42. JavaScript micro frameworks
- 43. JavaScript tooling
- 44. Log aggregation & indexing
- 45. Message buses without smarts
- 46. NuGet
- 47. Polyglot persistence
- 48. PowerShell
- 49. PSake
- 50. Vagrant

Assess
- 51. Gradle
- 52. jQuery Mobile
- 53. Logic-free markup
- 54. Open source BI/ETL tools
- 55. Riak
- 56. Sonar

Hold
- 57. Code in configuration
- 58. Cross-platform mobile toolkits
- 59. Enterprise service bus
- 60. Maven
- 61. VCS with implicit workflow



▲ New or moved
● No change

Platforms

Adopt
- 62. ATOM
- 63. AWS
- 64. Care about hardware
- 65. Communication between those responsible for hardware and software
- 66. KVM
- 67. Mobile web

Trial
- 68. Domain-specific PaaS
- 69. Heroku
- 70. Linux containers
- 71. Offline mobile webapps (just HTML5)
- 72. Private clouds
- 73. Tablet
- 74. Ubiquitous computing
- 75. Windows Phone 7
- 76. AppHarbor

Assess
- 77. Cloud Foundry
- 78. GPGPU
- 79. Hybrid clouds
- 80. Node.js
- 81. OpenSocial
- 82. Single threaded servers with asynchronous I/O
- 83. vFabric

Hold
- 84. Buying solutions you can only afford one of
- 85. GWT
- 86. Java Portal Servers
- 87. RIA
- 88. Treating VMs like physical infrastructure
- 89. WS-*
- 90. Zero-code packages

Languages

Adopt
- 91. Care about languages
- 92. HTML5
- 93. JavaScript as a first-class language

Trial
- 94. Clojure
- 95. CoffeeScript
- 96. Domain-specific languages
- 97. SASS, SCSS, and LESS
- 98. Scala

Assess
- 99. ClojureScript
- 100. F#
- 101. Functional Java
- 102. Future of Java

Hold
- 103. Google Dart
- 104. Logic in stored procedures

Techniques

Emergent design is one of the more advanced aspects of agile engineering practices, and therefore an area of active research & development. Such architectures should be driven by the underlying technical requirements of the system, rather than speculative planning for a future that may change. We have identified at least two facets of emergent design: the Lean software principle of last responsible moment, which mostly applies to greenfield projects, and finding & harvesting idiomatic patterns, which is more applicable to existing projects.

We recommend adopting **evolutionary architecture** as an alternative to traditional up-front, heavy-weight enterprise architectural designs.

Micro-services, often deployed out-of-container or using an embedded HTTP server, are a move away from traditional large technical services. This approach trades benefits such as maintainability for additional operational complexity. These drawbacks are typically addressed using infrastructure automation and continuous deployment techniques. On balance, micro-services are an effective way of managing technical debt and handling different scaling characteristics especially when deployed in a service oriented architecture built around business capabilities.

A decade ago when memory was at a premium, application servers made a lot of sense. They were popular and useful as a mechanism to run and manage multiple applications on a shared server or cluster. These days applications are more often run on separate physical or virtual servers and the need for an application server is reduced. Consider evaluating **server / application container end-of-life** within your organization, and only use one if you benefit from the added complexity.

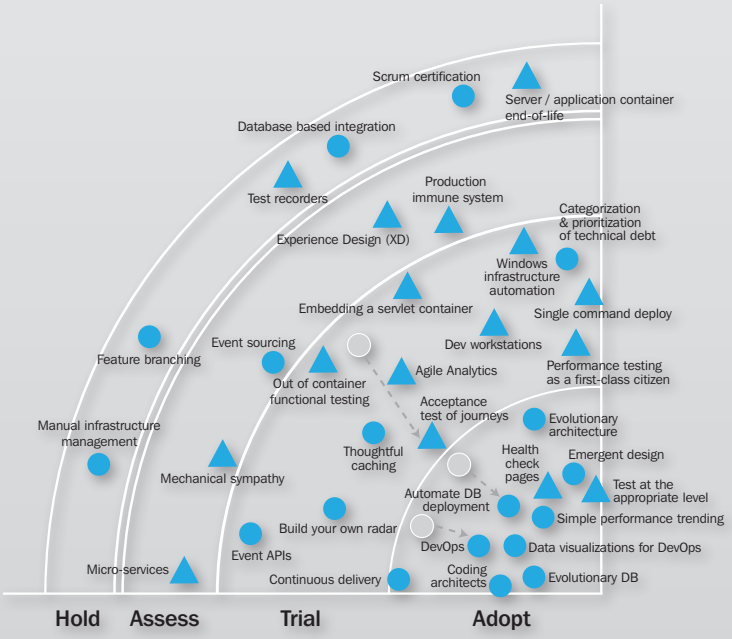
Embedding a servlet container, such as Jetty, inside a Java application has many advantages over running the application inside a container. Testing is relatively painless because of the simple startup, and the development environment is closer to production. Nasty surprises like mismatched versions of libraries or drivers are eliminated by not sharing across multiple applications. While you will have to manage and monitor multiple Java Virtual Machines in production using this model, we feel the advantages offered by the simplicity and isolation are significant.

With the popularity of embedded HTTP servers increasing, so has the technique of **out-of-container functional testing**. That is writing tests at the boundary of the system, using a mock container to provide both fast feedback and high coverage. Servers such as Jetty and tools like Plasma for the .Net platform can provide a significant reduction in the time it takes to run your test suite.

Experience Design (XD) is an example of ways in which agility must evolve to accommodate real-world constraints. We are always interested in finding innovative ways to incorporate what have traditionally been up-front exercises into practices like Continuous Delivery. XD is a ripe field for study.

There is a worrying trend that developers are becoming too distant from the hardware on which their code runs. Increasing virtualization and separation between development and operations makes this worse. In stark contrast some teams are writing code that leverages **mechanical sympathy** to get incredibly high performance from their software. The LMAX Disruptor is an open-source example in Java. For high performance cases like finance and Big data, getting closer to the metal can yield big returns.

Applying agile methods to data warehousing, business intelligence and **agile analytics** provides better return and improved business responsiveness. This is done by applying lightweight technologies like REST services to move data around in near real-time instead of batch updates. This allows information about customer behavior and application usage to be derived and responses incorporated within the applications for better user experience and data visualization.



Techniques *continued*

We have found adding simple **health check pages** to applications is incredibly useful. This allows people to quickly understand the health of an individual node. We often extend them to add metrics like the number of orders placed, error rates, or similar information. Using simple embedded web servers, even non-web based services can easily expose internal information over HTTP. By using microformats, these web pages can easily be scraped by other monitoring tools to become part of holistic monitoring.

A key step in the Continuous Delivery process is the ability to release software arbitrarily close to when the business wants it. The ability to do **single command deploy** relies on a complete set of activities that fall under the umbrella of Continuous Delivery including extensive automation of everything from build/test to scripted environment provisioning and deployment. We have found that adopting this as a goal tends to drive the automation and testing pre-requisites upstream into the rest of your organization.

Continuous Delivery techniques are shortening the "last mile" to get changes into production, allowing more frequent feature releases. A **production immune system** tracks changes as they are put into production, and automatically rolls back changes that have a negative effect on key metrics, such as revenue. Solid metrics, as well as automated A/B deployment, are required for this kind of aggressive rollback to be successful.

Automation is one of the core practices of Continuous Delivery. While companies are getting better at automating the management of infrastructure and environments, one commonly forgotten aspect is **infrastructure automation of development workstations**. This leads to huge gains in productivity by avoiding manually building specific environments and allows a seamless pairing environment. As with other parts of the environment, tools like Puppet and Chef can be used though they are not entirely necessary as the judicious use of platform packaging and language build tools can be sufficient.

Mature tools such as PowerShell, together with newer options such as Chef and Puppet, lead us to highlight **Windows infrastructure automation** on this edition of the technology radar. Manual configuration using a mouse and menu options is slow and leads to misconfiguration and "snowflake" machines in an unknown state. We recommend command-line tools for their clarity and scriptability.

We have long advocated for both static and dynamic code analysis tools to help glean information about your code base. As the focus of software development broadens because of the Continuous Delivery movement, **data visualizations of development and operations** with effective, actionable profiling and monitoring should be part of your technical stack as well.

Story level testing can lead to a focus on completing individual stories instead of coherent functionality. It tends to produce a large, hard to maintain test suite which runs slowly, delaying the feedback loop. The alternative is **user journeys**, which are groupings of user stories into sets of user interactions that provide value for both users and the business. Automating these into a suite leads to tests which hold their intent for longer periods of time and whose failure reveals a failure in the application's ability to deliver concrete value to its users.

The advent of behavior-driven design (BDD) testing frameworks like Cucumber, combined with browser automation tools like Selenium, has encouraged widespread use of acceptance testing at the browser level. This unfortunately encouraged doing the bulk of testing where the cost to run the tests is the greatest. Instead, we should **test at the appropriate level**, as close to the code as possible, so that tests can be run with maximum efficiency. Browser-level tests should be the icing on the cake, supported by acceptance and unit tests executed at appropriate layers.

While unit and acceptance testing are widely embraced as standard development practices, this trend has not continued into the realm of performance testing. Currently, the common tooling drives testers towards creating throw-away code and a "click-and-script" mentality. Treating **performance testing as a first class citizen** enables the creation of better tests that cover more functionality, leading to better tooling to create and run performance tests, resulting in a test suite that is maintainable and can itself be tested.

Test recorders seem invaluable as they provide a quick way to capture navigation through an application. However, we strongly advise against their regular use, as it tends to result in brittle tests which break with small changes to the UI. The test code they produce tends to be relatively poor and riddled with unnecessary duplication. Most importantly, test recorders tend to cut channels of communication between the test automation and development teams. When faced with an application that is difficult to test through the user interface, the solution is to have a critical conversation between the teams to build a more testable UI.

Tools

Polyglot persistence is the technique of storing data in various data stores based on efficiency and how that data is going to be used. Do not just use the default database, often an RDBMS, for all the needs of the application. Instead, ask questions like: Does session management data belong in the database or does it belong in a key-value store? Do relationships between customers and products belong in a graph database? Using **NoSQL** databases like MongoDB, Riak and Neo4J allows us to reconsider how data is treated, even with-in a single application.

Riak is a distributed key-value store that is schemaless and data-type agnostic. It can be put to good use in write heavy projects to store data such as sessions, shopping carts and streaming logs. The ability of the distributed cluster to self recover, distribute data across the cluster with tunable consistency and availability settings, do collision detection and resolve those if needed can be helpful in high availability environments and provide interesting solutions in the architecture.

With mobile applications on the rise, JavaScript size and performance is even more critical. **JavaScript micro frameworks** have emerged as a direct response to 'bloat' in some of the larger libraries. These small libraries do exactly one thing, such as DOM selection or MVC, and can be under one kilobyte in size. By combining a number of micro frameworks, developers can get exactly the functionality they need without the overhead of a larger library. Microjs.com hosts a collection of these micro frameworks, as well as a tool that can bundle them into a single library.

JavaScript is now established as a powerful, mainstream language that can be used in a variety environments both on client and server sides. As JavaScript codebases expand, more **JavaScript tooling** support becomes necessary, especially in the continuous integration and testing spaces. Tools like Backbone.js, SpineJS, JavaScriptMVC, Jasmine, JSTestDriver and HRcov are coming to the forefront. They are created by a vibrant community that continues to grow.

Frank is an open source library that allows functional tests for iPhones and iPads to be written in Cucumber and executed on a remote device. This fills an important niche in iOS development where acceptance test-driven development was previously cumbersome and awkward.

While MVC has been a staple of web development for the past few years, most libraries and frameworks fail to adhere to one of its most important principles: keeping logic out of the view layer. The consequences of not having **logic-free markup** include complex dependencies, difficulty testing and inability to reuse code. Recent DSLs like Mustache are available for many languages and platforms and have started to turn the trend. They allow editing in any tool, without extra requirements for language support and provide huge gains for UI development and overall application design.

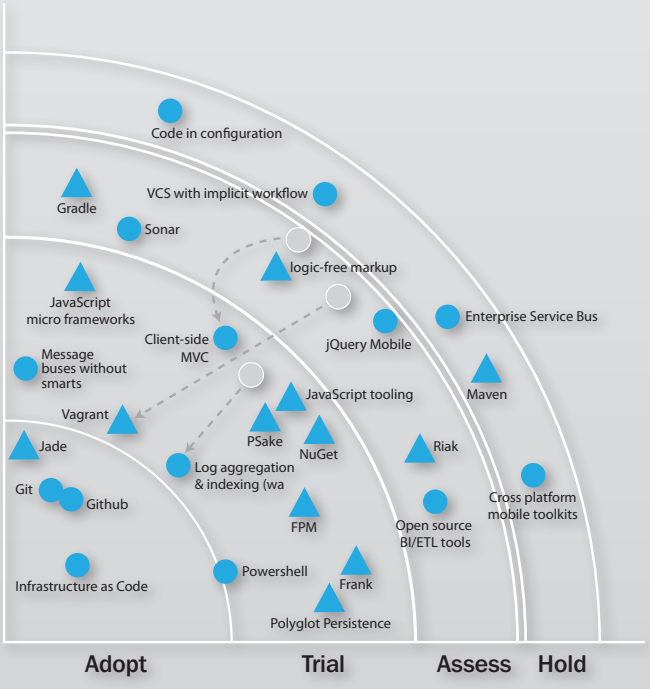
We continue to highlight **infrastructure as code**. This technique treats infrastructure configuration in the same way as code; checking configuration into source control, then carefully pushing changes out to the data center.

There are many advantages to using OS-native packages to deploy components and dependencies, however the tools which build native packages for Linux are not trivial. **FPM** is a useful tool which makes it easy to create RPM, DEB, or Solaris packages with a minimum of fuss.

Package management systems are a widely accepted practice for incorporating third party libraries. Tools such as RubyGems, Maven, APT, are available at both language and system level. **NuGet** is such a system for .Net platform. It consists of a Visual Studio IDE extension and a PowerShell module that opens the possibility for further improving build automation on the .Net platform.

PSake (pronounced 'sake' like the Japanese rice wine) is a build automation tool implemented in PowerShell. PSake provides a tidy syntax for declaring build tasks and dependencies without programming in XML. You also have access to all the features of PowerShell and the .NET framework from within your build scripts.

Maven has long been a staple of build automation in the Java space. However, given its lack of flexibility and support for automation best practices, especially in the Continuous Delivery domain, the use of alternatives such as Gradle should be considered.



Platforms

Node.js is just one example of a class of **single threaded servers with asynchronous I/O** that are seeing increased popularity. A traditional web or application server associates each incoming request with a thread until all the processing tasks associated with that request are complete and the response has been sent back. If any of those tasks involve I/O, the thread blocks while that I/O takes place. This approach can waste finite resources such as file descriptors and memory since each connection occupies a thread whether or not that thread is actually consuming CPU cycles. An alternative architecture is starting to emerge in implementations like Node.js (a JavaScript server running on Google V8), Nginx (an open source web server and proxy), and Webbit (a Java application server), that uses a single thread to serve many connections, processing all I/O asynchronously. These servers support orders of magnitude more simultaneous connections because each one consumes far fewer resources.

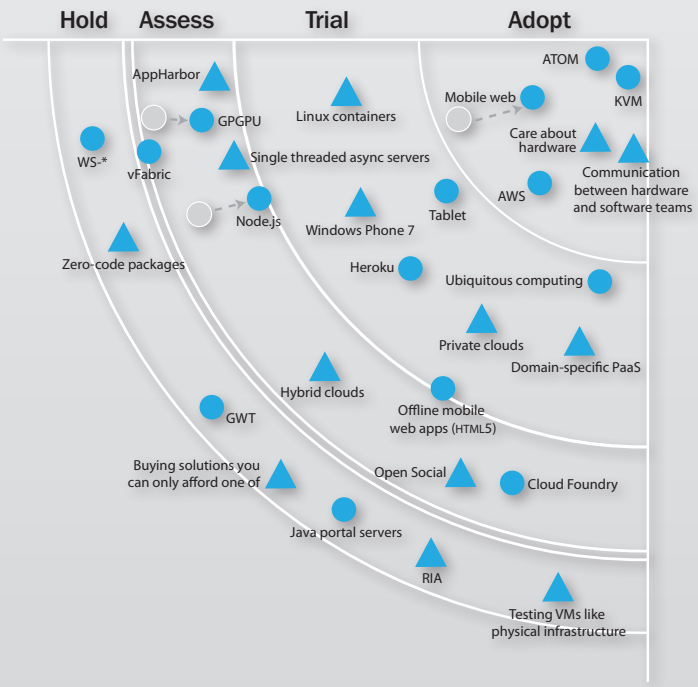
Because of concerns over privacy and security, or a need to repurpose existing hardware investments, many businesses are choosing to implement their own **private cloud**. There are are a variety of products, both open source and commercial for this purpose, but it should be noted that compute, storage, and network management are only the starting points for a useful private cloud. There are many services and processes that must be custom implemented to provide a cloud facility that rivals the public offerings from Amazon, Rackspace, or others.

Hybrid clouds describe a set of patterns that combine the best features of public clouds and private data centers. They allow applications to run in a private data center during normal periods then use rented space in a public cloud for overflow capacity during peak traffic periods. Another way to combine public and private clouds in an agile way is to use the elasticity and malleability of public clouds for developing and understanding an application's production characteristics, then moving it into permanent infrastructure in a private data center when it is stable.

While it can be all too easy to ignore geographical location of cloud-based services, for legal and technical reasons it can be a serious constraint when considering appropriate platforms. With the recently announced Brazil and Singapore regions, Amazon has made AWS-based systems more viable for people in areas previously poorly served by IaaS providers. In addition, they continue to add features to existing services, such as VPC. We remain confident in recommending **AWS** for those situations where flexibility in provisioning resources is key.

AppHarbor is a Platform as a Service (PaaS) offering for the .NET Platform using the same pricing model and structure pioneered by Heroku. It is a promising take on the deployment of .NET applications as it abstracts away most of the underlying configuration needs that come with the platform. It is maturing quickly and we expect it will see growing interest in time to come.

One style of virtualization that is particularly attractive for SaaS and PaaS implementations is the virtual container. **Linux containers** such as OpenVZ provide the isolation and management benefits of a virtual machine without the overhead usually associated with general-purpose virtualization. In the container model, the guest OS is limited to being the same as the underlying host OS, but that is not a serious limitation for many cloud applications.



Platforms *continued*

We find that many businesses are starting to build their own internal cloud deployment environments that can be easily replicated for development and testing environments. In many cases, provisioning is self-service, and with a single keystroke, developers can create a set of hosts that implement core enterprise assets and collaborating systems. In a sense, this is a **domain-specific PaaS** offered to internal customers.

Windows Phone 7 has surprised even some of the long time critics of Windows platforms. After many failed attempts, Microsoft has managed not only to produce a mobile operating system that provides a user experience on par with the other major contenders in the space but also the development support to go with it. Microsoft is making Windows Phone 7 a viable competitor and another choice for a more integrated experience in the corporate arena. Whether it will be able to change adoption trends remains to be seen.

OpenSocial is a specification that provides a standard way to share content between semi-trusted applications. While initially proposed for public facing social networking sites, it has possibly more potential within the corporate firewall, where the benefits of being able to share data and content between applications in a standard manner frequently outweigh the requirements of scale and security.

One of the principal mechanisms that allows agile software development to work is feedback loops. One common yet expensive broken feedback loop we have observed is the lack of **communication between those responsible for hardware and software**. The end result creates cost but not worth. You must view architecture holistically; neither hardware nor software has a full enough perspective to be successful alone.

While virtualization is on the rise, some organizations are **treating virtual machines like physical infrastructure**. We frown on doing a full operating system install for each VM or using VMs for load testing. Virtual machines can be cloned, snapshotted, and manipulated in ways physical machines cannot, and also have vastly different performance characteristics than physical hardware. VMs should be used with full understanding of their benefits and limitations, otherwise you can really get into trouble with them.

Many teams encounter problems that are caused by their test environment missing an expensive hardware component that is only present in production. While a pre-production environment in many cases cannot approach the scale of a production environment, all of its components should be present. We recommend not **buying solutions you can only afford one of**, such as SAN, firewalls or load balancers, as this prevents realistic testing anywhere but in production.

We have long been less than enthusiastic about **RIA** technologies such as Flash and Silverlight because of vendor lock in potential, anemic support for agile engineering practices, and potential for overuse. It seems even the large vendors are starting to agree with us. Now that modern versions of HTML handle most of the common cases that formerly required RIA, we feel that new projects must have enormous justification and careful strategic thought before using any of these technologies.

There are a number of enterprise software packages on the market that purport to offer flexible functionality with zero coding. This is certainly an appealing notion – that a non-technical business user could configure software to the unique requirements of any business without learning a programming language or hiring a professional software developer. However, it should be kept in mind that any change that affects the behavior of software in production, whether it is code, configuration, data or environments, could cause defects or failures in the business system. Writing code is only one step in a professional software production lifecycle. The need for repeatable analysis, testing, build, and deployment does not go away because the system is modified via a drag-and-drop interface instead of a high-level programming language. When evaluating a **zero-code package**, ensure that the the product supports these processes and that you have the necessary IT support structures in place to implement them.

We are reiterating our advice that given the progress and acceptance of simpler web-as-platform techniques such as REST and OAuth and the known issues with **WS-***, it should only be used cautiously.

Languages

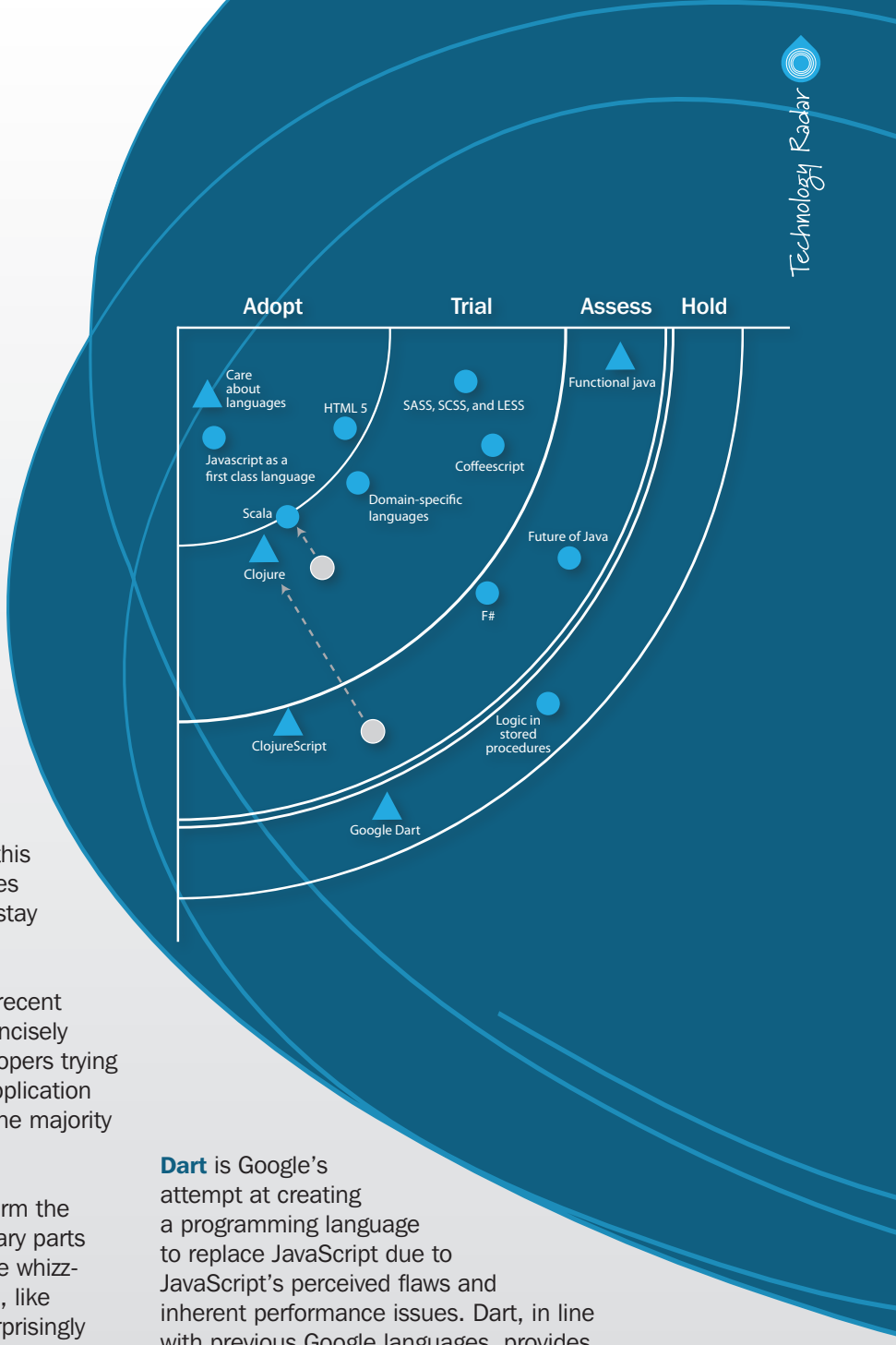
The industry is experiencing something of a renaissance in programming languages. ThoughtWorks thinks it is time to start assessing which other languages will help your organization while taking stock of the useful lifetime remaining for your current choices. You need to **care about languages**. Traditionally structured organizations with separate support teams may find skills constrain choice, DevOps offers a path forwards here.

Functional programming continues its slow but steady ascent into developer mind share and, increasingly, code bases. New languages like Clojure, Scala, and F# offer great new features. Now libraries such as **Functional Java**, TtotallyLazy and LambdaJ are back porting some functional language capabilities, particularly around higher-order functions and collections, into Java. We like this trend because it previews common capabilities of future languages yet allows developers to stay in their comfort zone.

Microsoft's **F#** continues to evolve, with the recent release of F# 3.0 beta. F# is excellent at concisely expressing business and domain logic. Developers trying to achieve explicit business logic within an application may opt to express their domain in F# with the majority of plumbing code in C#.

ClojureScript illustrates just how cross-platform the core of Clojure really is: they ported the primary parts to run on JavaScript. It is missing some of the whizz-bang features of Clojure on the JVM and CLR, like software transactional memory, but has a surprisingly high fidelity with its more sophisticated cousins. One interesting option afforded by ClojureScript is the ability to send data structures à la JSON using ClojureScript as the data structure. Because Clojure is a Lisp, this means that you can also send "real" code.

Rich experiences delivered via the web to desktops, tablets and mobile devices rely heavily on JavaScript, and we continue to recommend treating **JavaScript as a "first class" language** within your application. Developers should carefully consider how they structure, test, refactor and maintain JavaScript code, applying the same rigor as they would with any other programming language.



Dart is Google's attempt at creating a programming language to replace JavaScript due to JavaScript's perceived flaws and inherent performance issues. Dart, in line with previous Google languages, provides Java-like syntax and semantics that are intended to be more appealing than JavaScript's prototype-based nature. Reception within the browser-development community has been understandably cool and it remains to be seen if the language will become more widely accepted, though Chrome's continued rise and the search for alternatives like CoffeeScript may yet shift that balance.

References

LMAX Disruptor **http://martinfowler.com/articles/lmax.html**
Polyglot persistence **http://martinfowler.com/bliki/PolyglotPersistence.html**
JavaScript microframeworks **http://microjs.com/**
Frank – IOS testing **http://testingwithfrank.com/**
Functional Java **http://functionaljava.org/**

ThoughtWorks is a global IT consultancy

ThoughtWorks – The custom software experts. A company wholly devoted to the art and science of custom software. We make it, and we make our clients better at it. Our bottom line is to design and deliver software fast and predictably. Doing enterprise-scale software is tough, but the returns to those organizations that can deliver – on target – are tremendous.

ThoughtWorks' products division offers tools to manage the entire Agile development lifecycle through its Adaptive ALM solution™, comprised of Mingle®, Go™ and Twist®.

ThoughtWorks employs 1,800 professionals to serve clients from offices in Australia, Brazil, Canada, China, Germany, India, Singapore, South Africa, the United Kingdom and the United States. We lead the industry in rapid, reliable and efficient custom software development. When you need an expert partner to help you get ahead and stay ahead of the competition, get in touch.