

## Miles to go 4.0 ...

Arun Gupta is a technology enthusiast, avid runner, author of a best-selling book, globe trotter, a community guy, Java Champion, JavaOne Rockstar, JUG Leader, Minecraft Modder, NetBeans Dream Teamer, Devv4Kids-er, Docker Captain and works at AWS.

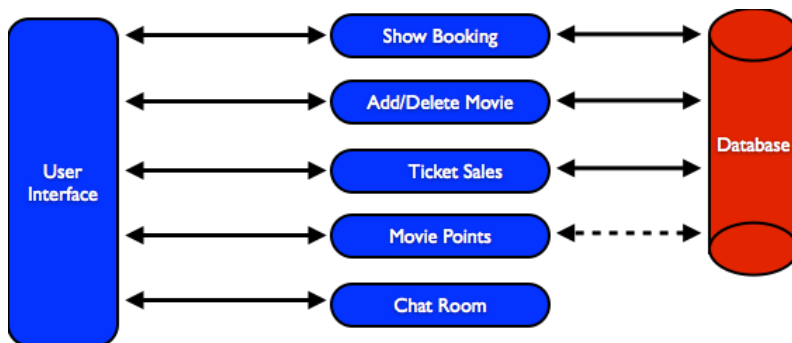
## Microservices, Monoliths, and NoOps

🕒 March 30, 2015   📁 microservices   🔗 javaee, microservices

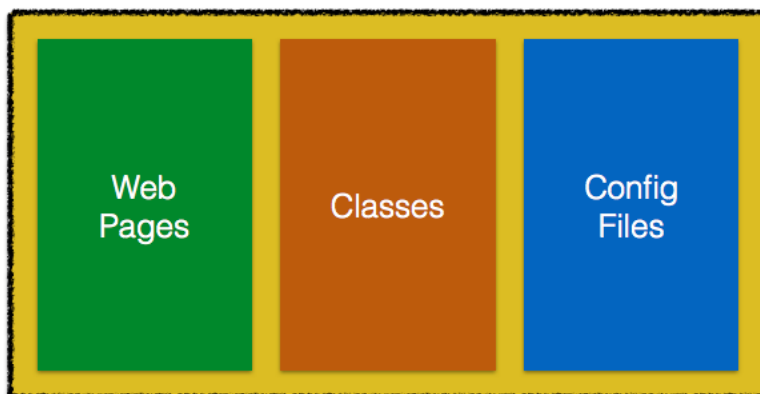
### Monolithic Applications

A monolith application, in layman terms, is where entire functionality of the application is packaged together as a single unit or application. This unit could be JAR, WAR, EAR, or some other archive format, but its all integrated in a single unit. For example an online shopping website will typically consists of customer, product, catalog, checkout, and other features. Another example is of a movieplex. Such an application would typically consist of show booking, add/delete movie, ticket sales, accrue movie points, and other features. In case of a monolithic application, all these features are implemented and packaged together as one application.

Movieplex7 is one such **canonical Java EE 7 sample application** and the main features are shown below:

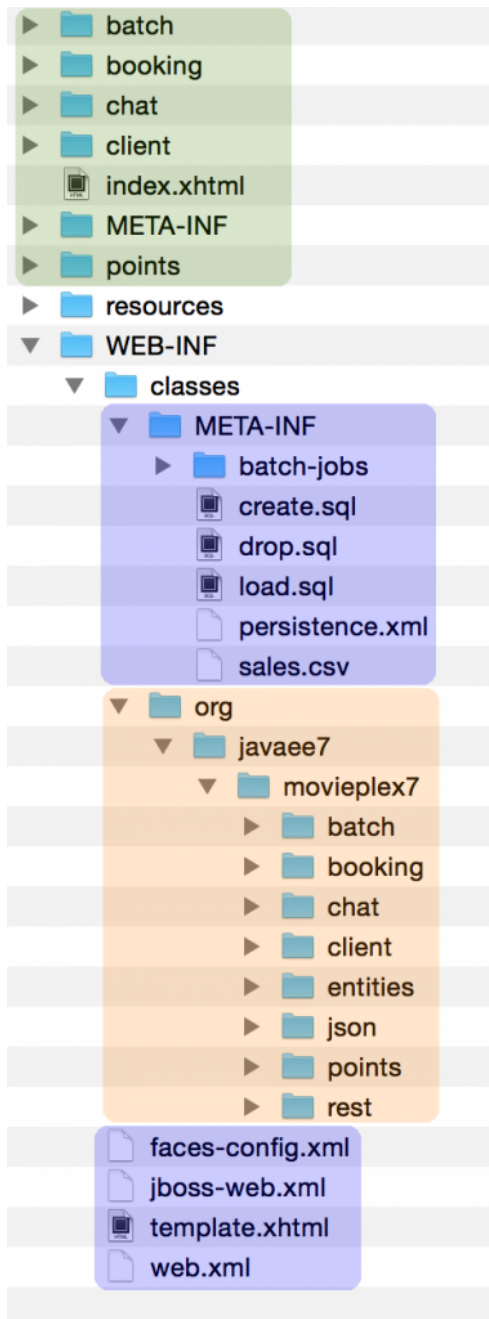


This application when packaged as a WAR would looks like:



The archive consists of some web pages that forms the UI. Classes implement the business logic, persistence, backing beans, etc. And finally there are some configuration files that define database connection, CDI configuration, etc.

More specifically, structure of the WAR looks like:



In this WAR structure, web pages are within the green box, all classes are within the orange box, and configuration files are within the blue box.

This application is somewhat modular as all the classes are neatly organized in package by different functionality. Web pages and configuration files follow a similar pattern as well.

## Advantages of Monolithic Applications

There are a few advantages of this style of application:

1. **Well Known:** This is how typically applications have been built so far. Its easy to conceptualize and all the code is in one place. Majority of existing tools, application servers, frameworks, scripts are able to deal with such kind of applications.
2. **IDE-friendly:** Development environments, such as NetBeans, Eclipse, or IntelliJ, can be easily setup for such applications. IDEs are typically designed to develop, deploy, debug, profile a single application easily. Stepping through the code base is easy

because the codebase is all together.

3. **Easy Sharing:** A single archive, with all the functionality, can be shared between teams and across different stages of deployment pipeline.
4. **Simplified Testing:** Once the application is deployed successfully, all the services, or features, are up and available. This simplifies testing as there are no additional dependencies to wait for in order for the testing to begin. Either the application is available, in which case all features are available, or the application is not available at all. Accessing or testing the application is simplified in either case.
5. **Easy Deployment:** Easy to deploy since, typically, a single archive needs to be copied to one directory. The deployment times could vary but the process is pretty straight forward.

## Disadvantages of Monolithic Applications

Monolith applications have served us well so far, and most likely will continue to work for some in the years to come. There are websites like Etsy that have 60 million monthly visitors and 1.5 billion monthly page views and are built/deployed as one large monolith. They have taken monoliths to an extreme where they are doing 50 deploys/day using a single large application. Unfortunately, most of the companies are not like that.

A monolithic application, no matter how modular, will eventually start to break down as the team grows, experienced developers leave and new ones join, application scope increases, new ways to access the applications are added, and so on. Take any monolith application that has spanned multiple years and teams and the entire code base will look like a **bug ball of mud**. That's how software evolves especially when there is a pressure to deliver.

Lets look at some of the disadvantages of monolithic applications

- **Limited Agility:** Every tiny change to the application means full redeployment of the archive. Consider the use case where only one piece of functionality in the application needs to be updated, such as booking or add/delete movie. This would require the entire application to be built again, and deployed again, even though other parts of the application has not changed. This means that developers will have to wait for the entire application to be deployed if they want to see the impact of quick change made in their workspace. Even if not intentional, but this may require tight coupling between different features of the application. This may not be possible all the time, especially if multiple developers are working on the application. This reduces agility of the team and the frequency by which new features can be delivered.
- **Obstacle for continuous delivery:** The sample application used here is rather small so the time it takes to rebuild and deploy the archive would not be noticeable much. But a real-life application would be much bigger, and deployment times can be frustratingly long and slow. If a single change to the application would require entire application to be redeployed then this could become an obstacle to frequent deployments, and thus an impediment of continuous deployment. This could be a serious issue if you are serving a mobile application where users expect latest cool new feature all the time.
- **“Stuck” with Technology Stack:** Choice of technology for such applications are evaluated and decided before the application development starts. Everybody in the team is required to use the same language, persistence stores, messaging system, and use similar tools to keep the team aligned. But this is like fitting a square peg in a round hole. Is MySQL an appropriate data store for storing graph databases? Is Java the most appropriate language for building front-end reactive applications? Its typically not possible to change technology stack mid stream without throwing away or rewriting significant part of existing application.
- **Technical Debt:** “Not broken, don't fix it” methodology is very common in software developed, more so for monolithic applications. This is convenient and enables to keep the application running. A poor system design or badly written code is that much difficult to modify because other pieces of the application might be using it in unexpected ways. Software entropy of the system increases over a period of time unless it is refactored. Typically such an application is built over several years with the team that is maintaining the code base completely different from the one that created the application. This increases technical debt of the application and makes it that much harder to refactor the application later on.

## What are Microservices?

The growing demand for agility, flexibility, and scalability to meet rapidly evolving business needs creates a strong need for a faster and more efficient delivery of software.

## Meet Microservices!

Microservices is a software architectural style that require functional decomposition of an application. A monolithic application is broken down into multiple smaller services, each deployed in its own archive, and then composed as a single application using standard lightweight communication, such as REST over HTTP. The term “micro” in microservices is no indication of the LOCs in the service, it only indicates the scope is limited to a single functionality.

We’ve all been using microservices for a few years already. Think about a trivial mobile application can tell you the ratings of a hotel, find out the weather at your destination, book the hotel, locate directions to your hotel, find a nearby restaurant, and so on. This application is likely using different services such as Yelp, Google Maps, Yahoo Weather API, etc to accomplish these tasks. Each of this functionality is effectively running as an independent service and composed together in this single mobile application. Explosion of mobile apps, and their support for the growing business demand is also highlighted by [Forrester’s four-tier engagement platform](#), and services are a key part of that.

Lets look at what are the characteristics of a microservice based application.

# Characteristics of Microservices

Lets look at the characteristics of an application built using microservices.

- **Domain Driven Design:** Functional decomposition of an application can be achieved using well-defined principles of [Domain Driven Design by Eric Evans](#). This is not the only way to break down the applications but certainly a very common way. Each team is responsible for building the entire functionality around that domain or function of the business. Teams building a service include full range of developers, thus following the *full-stack development* methodology, and include skills for user interface, business logic, and persistence.
- **Single Responsibility Principle:** Each service should have responsibility over a single part of the functionality, and it should do that well. This is one of the [SOLID](#) principles and has been very well demonstrated by Unix utilities.
- **Explicitly Published Interface:** Each service publishes an explicitly defined interface and honors that all times. The *consuming* service only cares about that interface, and does not, rather should not, have any runtime dependency on the *consumed* service. The services agree upon the domain models, API, payload, or some other contract and they communicate using only that. A newer version of the interface may be introduced, but either the previous versions will continue to exist or the newer services are backwards compatible. You cannot break compatibility by changing contracts.
- **Independently Deploy, Upgrade, Scale, Replace:** Each service can be independently deployed, and redeployed again, without impacting the overall system. This allows a service to be easily upgraded, for example to add more features. Each service can also scale independently on X-axis (horizontal duplication) or Z-axis (lookup oriented splits) as defined in [Art of Scalability](#). Implementation of the service, or even the underlying technology stack, can change as long as the exact same contract is published. This is possible because other services rely only upon the published interface.
- **Potentially Heterogeneous/Polyglot:** Implementation detail of the one service should not matter to another service. This enables the services to be decoupled from each other, and allows the team building the service to pick the language, persistence store, tools, methodology that is most appropriate for them. A service that requires to store data in a RDBMS can choose MySQL, and another service that needs to store documents can choose Mongo. Different teams can choose Java EE, NodeJS, Python, Vert.x, or whatever is most efficient for them.
- **Light-weight Communication:** Services communicate with each other using a light-weight communication, such as REST over HTTP. This is inherently synchronous and so could have some potential bottle necks. An alternative mechanism is to use publish-subscribe mechanism that supports asynchronous messaging. Any of the messaging protocols such as AMQP, STOMP, MQTT, or WebSocket that meet the needs can be used there. Simple messaging implementations, such as ActiveMQ, that provide a reliable asynchronous fabric are quite appropriate for such usages. The choice of synchronous and asynchronous messaging is very specific to each service. They can even use a combination of the two approaches. Similarly the choice of protocol is very specific to each service but there is enough choice and independence for each team building the service.

[Netflix](#) is a poster child for microservices and [several articles have been published on their adoption of microservices](#). A wide range of utilities that power their architecture are available at [netflix.github.io](#).

# Advantages of Microservices

- **Easier to develop, understand, and maintain:** Code in a microservice is restricted to one function of the business and is thus easier to understand. IDEs can load the small code base very easily and keep the developers productive.
- **Starts faster than a monolith:** Scope of each microservice is much smaller than a monolith and this leads to a smaller archive. As a result the deployment and the startup is much faster keeping developers productive.
- **Local change can be easily deployed:** Each service can be deployed independent of other services. Any change local to the service can be easily made by the developer without requiring coordination with other teams. For example, performance of a service can be improved by changing the underlying implementation. As a result this keeps the agility of the microservice. This is also a great enabler of CI/CD.
- **Scale independently:** Each service can scale independently using X-axis cloning and Z-axis partitioning based upon their need. This is very different from monolithic applications that may have very different requirements and yet must be deployed together.
- **Improves fault isolation:** A misbehaving service, such as with a memory leak or unclosed database connections, will only affect that service as opposed to the entire monolithic application. This improves fault isolation and does not bring the entire application down, just a piece of it.
- **No long term commitment to any stack:** Developers are free to pick language and stack that is best suited for their service. Even though the organizations may restrict the choice of technology but you are not penalized because of past decisions. It also enables to rewrite the service using better languages and technologies. This gives freedom of choice to pick a technology, tools, and frameworks.

Microservices may seem like a silver bullet that can solve significant amount of software problems. They serve a pretty good purpose but are **certainly not easy**. A significant operations overhead is required for these, and [this article from Infoworld](#) clearly points out.

*“with microservices, some technical debt is bound to shift from dev to ops, so you’d better have a crack devops team in place*

This is very critical as now your one monolith is split across multiple microservices and they must talk to each other. Each microservice may be using a different platform, stack, persistent store and thus will have different monitoring and management requirements. Each service can then independently scale on X-axis and Z-axis. Each service can be redeployed multiple times during the day.

## Microservices and NoOps

This imposes additional requirements on your infrastructure. These are commonly put together and called as **NoOps**. Essentially these are a set of services that provide a better process for deploying applications, and keep them running.

- **Service replication:** Each service needs to replicate, typically using X-axis cloning or Y-axis partitioning. Does each service need to build their logic to scale? For example, Kubernetes provide a great way to replicate services easily using [Replication Controller](#).
- **Service discovery:** Multiple services might be collaborating to provide an application’s functionality. This will require a service to discover other services. It could be tricky in a cloud environment where the services are ephemeral and possibly scale up and down. Resolving the services that are required for a service is thus a common functionality for all other services. Services need to register with a central registry and other services need to query this registry for resolving any dependencies. [Netflix Eureka](#), [Etcd](#), [Zookeeper](#) are some options in this space ([more details](#)).
- **Resiliency:** Failure in software occurs, no matter how much and how hard you test it. The key question is not “how to avoid failure” but “how to deal with it”. This is all the more prominent in microservices where services are distributed all over the Internet. Its important for services to automatically take corrective action and ensure the user experience is not impacted.

Michael Nygard's book [Release It!](#) introduces Circuit Breaker pattern to deal with software resiliency. Netflix's [Hystrix](#) provide an implementation of this design pattern ([more details](#)).

- **Service monitoring:** One of the most important aspects of a distributed system is service monitoring and logging. This allows to take a proactive action, for example, if a service is consuming unexpected resources.

## Refactoring into Microservices

Microservices also does not mean you need to throw away your existing application. Rather in majority (99.9%?) cases, you cannot throw away the application. So you've to build a methodology on how to refactor an existing application using microservices. However you need to bring your monolith to a stage where it is ready for refactoring. As [Distributed big balls of mud](#) highlight:

“ *If you can't built a monolith, what makes you think microservices are the answer?* ”

Refactoring may not be trivial but in the long terms this has benefits which is also highlighted in the [previously quoted article from Infoworld](#):

“ *Refactoring a big monolithic application [using microservices] can be the equivalent of a balloon payment [for reducing technical debt] ... you can pay down technical debt one service at a time* ”

Functional decomposition of a monolith is very important otherwise it becomes a distributed monolith as opposed to a microservice based application.

## Future Blogs

A subsequent blog on [blog.arungupta.me](http://blog.arungupta.me) will show how to refactor an existing Java EE application using microservices.

Some more questions that would be answered in subsequent blogs ....

- How is it different from SOA?
- Is REST the only way to exchange data? What messaging protocols?
- Does microservices simplify/require CI/CD?
- How is it related to Containers and DevOps? Are containers required to run microservices?
- Are there any standards around microservices?
- Are we pushing the problems around to orchestration?
- What roles does PaaS play to enable microservices?
- How can existing investment be leveraged?
- [Microservices Maturity Model](#)

Be Sociable, Share!



Tweet

Like 25

G+

Share

110

Related posts:

1. **Gilt and Microservices: Why and How**
2. **Microservice Design Patterns**
3. **Microservices with JBoss EAP 6 Reference Architecture**
4. **WildFly Swarm: Building Microservices with Java EE**
5. **JavaOne Cloud, DevOps, Containers, Microservices etc. Track**

## *19 thoughts on “Microservices, Monoliths, and NoOps”*



Richard

March 30, 2015 at 6:56 pm

It would be useful to address the challenges of managing transactions across a suite of micro services especially given the trend of exposing the service interfaces as REST API's to abstract the different technology stacks used by the different services.

This is especially important if you are looking to move from a single large Java EE application that relies of the Java EE framework to ensure interactions between the different services/modules packaged in the application and external services such as DB's and Message Queues are transactional.



Dennis

March 31, 2015 at 4:13 am

Great article – Thank You

Binh Thanh  
Nguyen

March 31, 2015 at 10:52 pm

Thanks, nice post. Waiting for a series of posts.

April 1, 2015 at 6:06 am



John  
Allspaw

This is a well-written but quite misguided article, and points to what I think is one of the largest language deficits our industry has.

I can, if pushed, describe Etsy's architecture as using microservices. I can do this because frankly the terms are underspecified enough and context-dependent enough to be meaningless in the current industry dialogue to give pros and cons around.

How can you rectify the logical contrast between the disadvantages listed in "Obstacle for continuous delivery" and still describe Etsy's architecture as a monolith that yet engages in continuous delivery? There are more paradoxes in the topic, this is just one.

The patch that I will submit to this article is that there is still, frustratingly, an insufficiently explored map of context.

I will paste this once again, here, that I mentioned to Adrian Cockcroft:  
(<https://gist.github.com/jallspaw/bc60f27c38a2d9009f34>)

1. Monolithic applications and architectures can vary in their monolithness. This is an under-specified description.
  2. Microservice applications and architectures can vary in their microness. This is an under-specified description.
  3. Microservices and monolithic architectures have both benefits and disadvantages.
  4. Organizations will exploit those benefits while working around any weaknesses.
  5. Success of the business is a large influence on the exploitation of benefits and implementation and costs of workarounds.
  6. All benefits and work arounds are context-sensitive. Meaning that they are both technically and socially constructed by the organization that navigates them.
  7. Path dependency is a thing. History matters and manifests in these architectural decisions and evolution in an organization.
  8. Patterns exist to inform practice, not dictate it. Zealous adherence to an architectural pattern brings peril when it is to the exclusion of cultural context in actual practice.
  9. Architectural patterns will expand, contract, evolve, and change to fit the trade-offs that an organization perceives it has to make.
- "The danger for a movement that regards itself as new is that it may try to embrace everything that is not old." – Kim Vicente





Aaron

April 1, 2015 at 7:12 am

@John Allspaw

“How can you rectify the logical contrast between the disadvantages listed in “Obstacle for continuous delivery” and still describe Etsy’s architecture as a monolith that yet engages in continuous delivery? There are more paradoxes in the topic, this is just one.”

I see no paradox. The author never suggests that the two are incompatible – just that one is an obstacle when attempting to achieve the other.

I.e. the author is merely saying that monoliths make these things harder to achieve.

I think most people, author included, will take the 9 points you mention for granted. I’m not sure it’s (the case for micro over mono) in any way meant to be taken absolutely – nor do I think that anyone in their right mind would do so.

The reality is that most of the world is monolithic, and most of the time we’re making things harder for ourselves than we need to.

John  
Allspaw

April 1, 2015 at 7:29 am

Aaron:

I would love for that to be the case -> “(the case for micro over mono) in any way meant to be taken absolutely” I’m sensitive to how binary/absolute I’ve seen the topic elsewhere. So that is my sensitivity: noticing strong assertions in the article that fly in the face of the reality, which is a lot more nuanced and context-specific than is acknowledged here.



Aaron

April 1, 2015 at 8:05 am

@John:

In that case I do sympathise.

Thinking about it, I’ve been suffering recently with people who think that having a Jenkins box means they do CI – even though they only integrate feature branches every 3 months or so (just before the release).

I suppose that assuming common sense will be used is more frequently dangerous than it should be :s



★  
arungupta

April 1, 2015 at 5:50 pm

@John

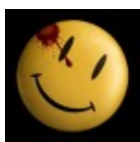
I agree with the nine points mentioned in your article. This article is no way promoting microservices or monolithic applications, its just highlighting the differences between the two. Later are a given reality in the world today, at least to the majority of developer and customers I talk to. Some are migrating their monoliths to take advantages of \*ilities (reliability, scalability, flexibility, agility, etc) offered by microservices.

Etsy is just one organization that is using monoliths and continuous delivery very effectively. And I know of other orgs that are using that combination as well. Monolith and CD are neither directly or inversely proportional to each other IMHO, its rather difficult to achieve CD with monoliths, primarily because of the size of change, sphagetti code, and lack of testing. In that sense, microservices do enable and simplify CD.

OK?

@Aaron

Agree with your point “having a Jenkins box means they do CI” is very misleading. People forget about the multiple times checkin during the day, or at least once a day.



Mark Little

April 2, 2015 at 12:29 am

@Richard I agree with you about transactions (though also things like workflow/coordination of running tasks is important, as well as others). However, as for transactions, there's a lot of relevant work that has been done over the years on loosely coupled transactions (search for extended transactions, for example). Then there's quite a bit of work that has happened in the last decade on REST and transactions, both for extended (compensation based) and ACID transactions, which is relevant here.



Mark Little

April 2, 2015 at 12:31 am

@John I don't think microservices are a global panacea, as some would like us to believe. Could go into details here, but easier to reference an interview I did with InfoQ a while back.

<http://www.infoq.com/news/2015/02/special-microservices-mark-little>

Pingback: [Microservice Design Patterns - Miles to go 2.0 ...](#)

Pingback: [Microservice Design Patterns | Voxxed](#)

Pingback: [Microservices with JBoss EAP 6 Reference Architecture - Miles to go 2.0 ...](#)

Pingback: [Top 3 DevOps Practices for Operational Stability - DevOps.comDevOps.com](#)

Pingback: [Monolithic to Microservices Refactoring for Java EE Applications - Miles to go 2.0 ...](#)

Pingback: [Monolithic to Microservices Refactoring for Java EE Applications | Social Marketing by WordPress](#)

Pingback: [Microservice Design Patterns | JAVA](#)



January 31, 2017 at 9:07 pm

I'm 2 years behind but delighted to read. Are there any disadvantages using microservices architecture ?

Aslam  
Sayyed



Pranab Das

June 20, 2017 at 8:32 pm

Arun, you may not recall me. However, I thought you were more into enterprise class software building where sharing, scaling with optimal resource utilization, pooling with statelessness are most important. All these are not possible with microservices.

While there are areas of applicability of microservices, through your blog are you not encouraging people to stay away from enterprise class applications?

In fact, if someone else wrote the same thing, it would be ok but from you, being an evangelist of JavaEE, it looks like optimally-sized stateless sessionbean pools are

no more relevant. Horizontal scaling, which would make more sense with enterprise class, may not be necessary (unless your microservices are so big that they are akin to enterprise class apps). Even some of the patterns like Value List Handler would no more be necessary.

Considering the above, I am giving my humble analogy. Let there be a restaurant, where there are two discernible classes of service: waiters and cooks. If we build the app as enterprise class, we would use clear interface of separation between the two services. Now suppose the restaurant caters to 100s of varieties cuisines, each of which is ordered by only a limited few. Here, we may require no waiters but more cooks. In fact, cooks may double up as waiters and take/serve orders.

IMHO, microservices is not utopia: one should not go by buzzwords and look in depth for suitability.