Learn how to architect cloud-ready applications that are easier to deploy, maintain, and scale. From designing for read-only file systems to architectural patterns like event sourcing, investigate approaches that make it easier for your applications to run on cloud-based computing platforms.

Cloud computing is an important tool in any architect's toolkit. It can allow you to prototype and launch new projects without having to worry about server configuration, it can cut the cost and reduce the risk of managing your servers, and it can allow for easy scaling, with the ability to spin up extra instances whenever you need them.

Frequently used acronyms

- SSL: Secure Sockets Layer
- TPS: Testing Procedure Specification
- UUID: Unique user ID

However, there are a number of architectural issues you need to consider when you start to develop applications for the cloud. This article highlights some of the considerations when you architect your first cloud-based application.

Benefits of cloud computing

Before jumping into recommendations on architecting for the cloud, it's worth spending a bit of time revisiting the benefits that a cloud-based architecture can provide:

- **Time to market.** If you have a new application to launch, you can focus on developing features rather than provisioning infrastructure. There is also no lead-time for server purchasing, setup, and configuration. With experience, you could code a proof of concept in the morning and have it running on a cloud server the same afternoon.
- **Scale.** Often, you don't know what kind of load your applications will encounter. You then have to walk the thin line between overprovisioning for load that might never materialize and underprovisioning, causing performance problems until you can purchase, configure, and
deploy additional hardware. With cloud-based infrastructures, you can launch a minimal set of instances and automatically scale the number of instances if the load increases.

- **Flexibility.** With auto-scaling in a cloud-based infrastructure, it is possible to dynamically change the number of server instances. So, if you have a substantial difference between peak loads (times of the day, when special events launch, when ads run, and so on) and average loads, you don't have to provision and pay for enough instances for peak load 24x7.

- **Simplicity.** Architecting and deploying scalable applications in-house require you to handle a lot of complexity. You need to research, select, and configure load balancers; develop solutions for scripting the reconfiguration of IP addresses on hardware failure; design scalable messaging-based e-mail services; and work out how to implement backup and disaster recovery capabilities for critical data. Most cloud providers include proven solutions to these problems that you can use.

- **Easy testing.** When you use a cloud-based infrastructure, you can easily spin up additional instances for either functional or load testing, so you don't have to keep expensive hardware available 24x7 for running tests.

- **Initial cost.** With no capital costs up front, cloud computing also reduces the cost of getting a new project to market, and the hosting costs only become significant if the application is popular. That is a high-quality problem to have.

- **Recovery.** Because of the way you have to architect and develop applications for the cloud, it's usually much easier to handle backing up important data and automated recovery from hardware or network failures.

**Architecting for the cloud**

For many architects, a cloud-based application is their introduction to designing effectively for scale, as there are good reasons to use the cloud even for smaller-scale applications. There are a number of considerations when scaling out an application that may not be intuitive to architects used to maximizing performance on a single server.

**Scaling up vs. scaling out**

The first time you develop an application that becomes popular, the initial response is usually to scale up. Add more RAM. Increase the processor speed. Upgrade to faster hard disks. Implement caching for slow queries or common calculations. These are all reasonable techniques, but if your application is ever to scale beyond a certain point, you're going to need to go beyond scaling up and design for scaling out across multiple servers. Doing so often requires substantial changes to the application's architecture. The very nature of cloud applications means that you should architect for scaling out from the start.

Also, when you architect for the cloud, it is important not to focus on performance micro-optimization. Rather than putting in a lot of complex caches and performance tweaks, it is much better to keep the design simple and make sure you can scale out the application. Servers are generally much cheaper than developers, and performance micro-optimizations usually aren't worth the penalty in developer time and reduced application maintainability.

**Key considerations for scaling out**

When an application has to scale out, focus on five architectural approaches:
• Minimize mutable state.
• Evaluate NoSQL data stores.
• Create asynchronous services.
• Automate deployment.
• Design for failure.

Minimize mutable state

Perhaps the most important pattern for increasing horizontal scalability in the cloud is the minimization of mutable state. It is the reason that functional programming languages are becoming so popular and that patterns like event sourcing (see Related topics for a link to more information) are being more frequently applied. The problem with shared mutable state (that is, variables that are shared across the application and can be changed over time) is that it plays havoc with scalability. Multiple servers and processes trying to update the same variables at the same time result in deadlocks, time-outs, and failed transactions. Three places you will want to minimize or eliminate mutable state in your application are on the web servers, in the application, and (possibly even) in the database.

Immutable file systems

With cloud applications, there is no guarantee that any given instance will be durable. If you have three web server instances all handling requests, any given instance could potentially go down at any time. Given that, you only want to store information on the local file system that you can afford to lose if the server instance fails.

If users are uploading files (whether profile pictures or TPS reports) you want to make sure to upload them to a redundant, remote file system—not just to the hard disk on the web server they are accessing. If you want to log information that you really need access to even if a server instance goes down, you might want to save it to a NoSQL store like Cassandra rather than persisting it to the local hard disk on any given web server.

Immutable applications

Shared mutable state is particularly useful in object-oriented programming, but it’s important to minimize the amount of mutable state and think carefully about the implications when scaling your cloud-based applications.

Eric Evans included some great code-level patterns in Domain Driven Design for minimizing mutable state, such as making your (mutable) entities as thin as possible and using (immutable) value objects for as much of your application state as possible. For example, a user is a mutable entity, but you might use an immutable value object to represent the user’s address, just changing the address that you point to for the user if he or she moves. In general, favor value objects and immutable values over mutable state, especially for state that might need to be updated by many parts of your cloud-based application on a regular basis.

Also, think carefully about your caching strategies. If information is specific to a single instance, a local cache is fine, as you can always re-create it if the instance stops functioning. But if the information might be required by users accessing another web server, you either want to pull
the information from a shared persistent store each time or use a distributed cache that handles updating the cache across multiple web server instances.

**Immutable data stores**

At first it might sound like an oxymoron, but there can be real benefits to taking an event sourcing-influenced approach to database design for cloud-based applications. A common problem as you try to scale applications is write contention in the database. A change to a purchase order could affect hundreds of order items and associated purchase orders for vendors. Naturally, any such change must be wrapped in a transaction to ensure that the database is never left in an invalid state, with only a subset of the changes having been made. But as you have more and more users updating transactions that affect related orders, pretty quickly you can get to a point where all of your transactions are failing because of optimistic locking conflicts.

One approach to this issue is to minimize mutable state in the database. Instead of having a user with mutable values for first name, last name, and other profile fields, you have a single record in the database with a unique identifier (for example, ID, e-mail address, UUID) but no additional state. You then have a separate table for `ProfileUpdateEvents` where you store an immutable event for every update with an associated time stamp. So, if a user updated his or her home city, there will be a `ProfileUpdateEvent` where the field is `home_city` and the new value and the date/time of the update is stored. To get a user's home city, you then just go through all of the `ProfileUpdateEvents` until you find the most recent time stamp that updated the field. This process gives you a system in which write contention disappears and you resolve any conflicts either on read or on an automated schedule, allowing for much greater write scalability in your application.

Of course, you now have an issue with read performance. Having to perform operations on a large collection of events just to get the state of a single entity isn't efficient, especially for applications for which read scalability is a concern. But that's an easy problem to solve: Re-create the additional fields in the user table, but instead of being the single authoritative reference for the state of the user, the fields are simply a cache of the state of the user at a certain point in time. Then, depending on your business rules, you can write scripts to update the cache of state in the user table based on the `ProfileUpdateEvents`, resolving any write conflicts based on your business rules. Event sourcing is worth considering if you have an application for which write contention is likely to be a real barrier to scaling your cloud-based application.

**Evaluate NoSQL data stores**

Another approach to managing write contention in the database is to evaluate the possibility of using NoSQL data stores for some or all of your application data. For example, Cassandra is designed specifically to provide linear write scalability across multiple nodes for storing huge quantities of data. CouchDB has excellent master-to-master synchronization across multiple nodes, and MongoDB has the concept of "counter" fields, allowing for asynchronous fire-and-forget updates to counter fields so that you don't have to worry about write contention if you're just updating the number of views on a post on a regular basis.
Many NoSQL stores also provide MapReduce capabilities, allowing for efficient access to predefined queries. If you're running known queries against large sets of data, they can provide a scalable solution as the amount of data you are working with grows.

Create asynchronous services
Another important architectural approach is to offload work from your main application servers by creating separate asynchronous services. There is no need for you block a thread and make your site visitors wait for a response while waiting to synchronously send e-mails or run reports. It is much better to break those kinds of tasks into separate services that can run on different servers in the cloud. That way, you can scale them independently from your main application. You don't have to block a thread on a web server while waiting for tasks to finish, and your users don't have to wait for those processes to finish to get a response saying that you're working on it and will notify them when the process is done.

You typically connect to your asynchronous services using a transport mechanism with some degree of guaranteed ability to deliver, such as a message queue. It's also important to use network rather than interprocess communication. Generally, run these services on separate servers; you certainly want to be able to run them on separate servers if you so choose. You can also use other architectural approaches, such as treating a database as a blackboard for sharing information between the main application and the various services that perform the separate asynchronous services.

Automate deployment
Finally, it is important that you automate your cloud deployment process. You need to have base machine images for the various roles (web server, e-mail server, database server, and so on) and to use tools like Chef or Puppet to automatically configure instances. You're also going to want to use some kind of build system for automatically deploying your code.

Monitoring for cloud-based applications is also an important concern. Set up monitoring so that you know when instances go down or are heavily loaded so that you can automate the process of spinning up more or fewer instances depending on how your application is performing.

Be sure to have automated, scripted re-mapping of IP addresses to handle cases when a data center goes down. Also, look into how your cloud provider supports SSL connections, mapping them to various instances. It's generally good practice only to use SSL on pages that absolutely require the transport security that the protocol provides. SSL requests carry overhead, so you only want to pay that premium when doing so adds business value.

To learn more about automating deployment and the growing DevOps movement, have a look at the excellent book on continuous delivery by Jez Humble and David Farley in Related topics.

Design for failure
It is particularly important to consider design for failure for your cloud-based applications. You should think through your doomsday scenario, looking at your single points of failure and what you would do in the event they failed.
However, it is equally important to decide what level of failure is acceptable. If a data center goes down, maybe it is okay if you can only provide Read access to your servers for a few minutes until you've promoted another database to "master." Maybe it is okay that if a server fails, a user will have to log in again or will lose his or her preferences or shopping cart. It is important to compare the cost of engineering for failure tolerance with the business value of that failure tolerance to make sure you don't over invest in providing redundancy for information that is not sufficiently important to your business.

**In conclusion**

Cloud-based applications provide many benefits, including getting to market more quickly and scaling more easily. In addition, the initial cost is lower, and well-architected cloud-based solutions typically provide much better disaster recovery capabilities. However, to take advantage of cloud-based deployments, you must architect in a way that is consistent with the cloud, generally assuming that you will be working with read-only local file systems and storing any important data either in a database or in a shared, persistent file system. Also, minimize mutable state in your applications and consider NoSQL data stores to improve the scalability of your applications. Look at how you can off-load any heavy processing to asynchronous services that could be moved to separate servers. Finally, automate your deployment process and to design your applications for failure to ensure that you get acceptable failure performance characteristics from your cloud-based application.
Related topics

- Learn more about event sourcing, not just knowing the state of an app but also how it got there, from Martin Fowler's website.
- In *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (by Jez Humble and David Farley, Addison-Wesley, 2010), you meet the "deployment pipeline," an automated process for managing all changes.
- "Open Source in the Cloud" (December 2010, Sys-Con Media) provides a thorough basic examination of the considerations and issues of using open source technologies in the cloud.
- Grace Walker's developerWorks article *Revolution in the air: Fundamentals of cloud computing* provides a good introduction to cloud computing. Another excellent resource for intro-level cloud technology knowledge is the series on service models PaaS, IaaS, and SaaS.
- In the developerWorks cloud developer resources, discover and share knowledge and experience of application and services developers building their projects for cloud deployment.
- See the product images available for IBM SmartCloud Enterprise.
- The next steps: Find out how to access IBM SmartCloud Enterprise.
- Learn more about and download Chef, an automated, open source systems integration framework.
- Puppet can help you build large cloud-based applications.