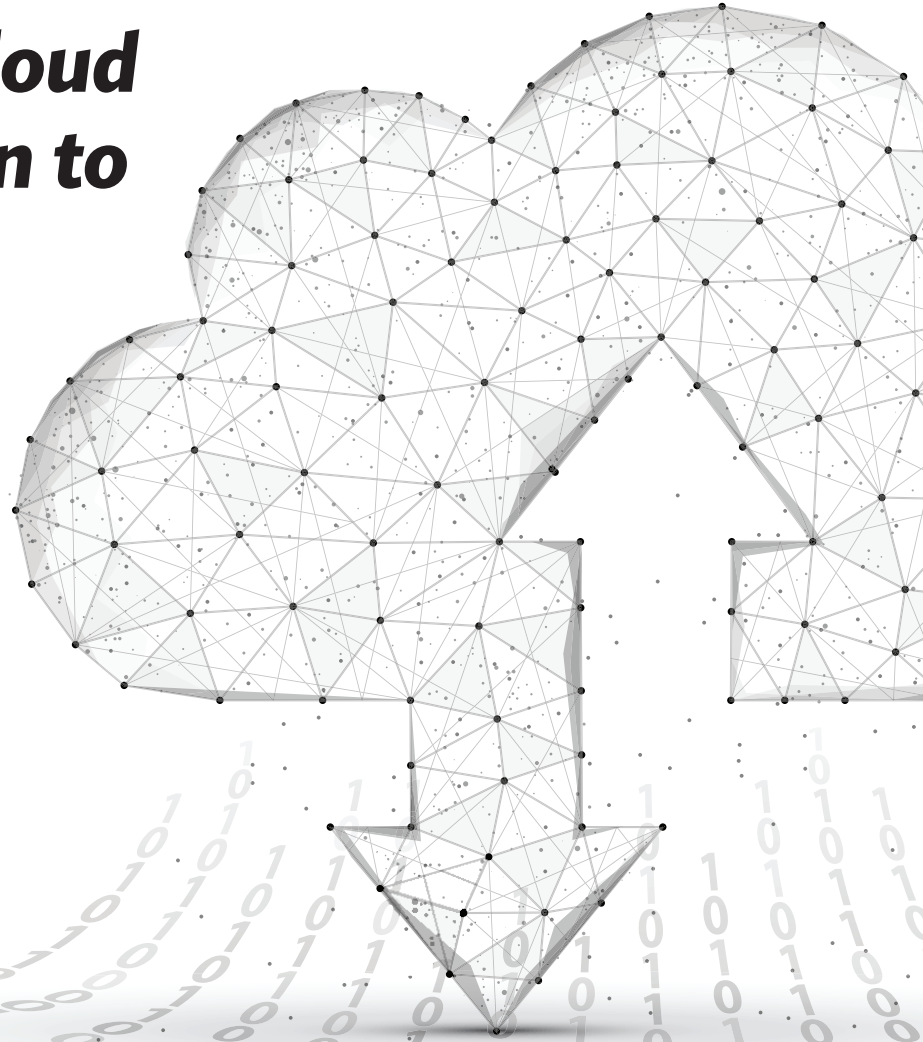
Coming Down from the Cloud:

A Concise Explanation of Cloud Computing and Introduction to Systems Architecture

Nicholas Schmidt, CIPP/US



iapp

Coming Down from the Cloud:

A Concise Explanation of Cloud Computing and Introduction to Systems Architecture

The mastery of parallel processing opened many exciting new possibilities, lowering the time necessary to solve complex computational problems while simultaneously raising the amount of computational power that can be leveraged to solve such problems. One of the developments to arise out of parallel processing technology is cloud computing, a proposal for large pools of available processors that could be borrowed by users in processing a large job then released to other users after the job is finished in a sort of "rental car" approach to processing. This was tremendously exciting for the computing industry as it meant that tasks that once required an enormous investment of processing time and could monopolize an individual machine for hours, weeks or months could now be completed quickly without having to individually purchase and maintain the necessary infrastructure. Many companies moved to capitalize on the excitement and billed their applications as using "the cloud," such that the term became a buzzword, fell out of favor and has now faded from marketing or promotional materials.

This is unfortunate because, with the rise of a subscription model of software distribution rather than the traditional model where software was sold as a good, more and more modern applications can rightfully lay claim to using "the cloud." Cloud applications have become ubiquitous in modern life even if they no longer advertise themselves as such; examples of popular cloud products include Smartsheet, Google Drive, Dropbox, YouTube, Gmail and GoDaddy.

In this third installment in a series of white papers, we'll introduce some of the common terms used when describing cloud applications and discuss some basic principles of software architecture, the abstract organization of software that are integral to a functional and secure cloud solution.

Interpreting "the cloud"

Here are some useful terms that will help you get down to brass tacks and identify what is being offered when assessing a "cloud" application:

"The cloud" — "The cloud" was originally supposed to refer to a pool of hardware resources that can be drawn on by users, usually over the internet. This includes processor cores that can be drawn upon to do computational work (traditional cloud computing), hard drives and distributed databases that can be drawn upon to hold information and documents (cloud storage), and other assets. Due to overuse, "the cloud" has

now come to mean any application (including software) or infrastructure hosted by the offeror and distributed to clients online using a licensing model.

_____ as a service (_aaS) — Whatever fills the blank — infrastructure (IaaS), software (SaaS), platforms (PaaS), functions (FaaS, i.e., platforms that can be used to create and maintain custom applications, and run them online), mobile backend (MBaaS or BaaS, provides the complicated infrastructure functionality for mobile app developers), etcetera — is what is being offered by the cloud vendor. For example, a SaaS model offers a software application to users over the internet.

Examples of _aaS applications:		
IaaS	Cloud storage applications	Google Drive, Dropbox
SaaS	Software applications	Smartsheet, Google Docs
PaaS	Hosting applications	GoDaddy
FaaS	Application development platform	Microsoft Azure Functions

Private versus public versus hybrid cloud — These terms refer to how the cloud product is offered. A private cloud is operated for the limited benefit of one organization, regardless of whether that organization manages it. A public cloud is offered to the public, either freely or by a subscription model. A hybrid cloud is a cloud service that crosses both major types or incorporates some of the smaller subtypes. Smaller subtypes of clouds include community clouds (semi-private clouds shared by a few organizations), distributed clouds (clouds composed of resources in different locations), multiclouds (combinations of several different cloud services into one cloud), and "big data" clouds (a buzzword of buzzwords, just cloud storage for large datasets).

Data or source code escrow — Occasionally, the licensee of a cloud computing service will request that the application's source code or the licensee's data be held in and regularly backed up into escrow by a third party to protect the application and its source code from abandonment by the provider or the licensee's data from catastrophic loss. Escrow functions just like any other regular database backup, is not uncommon, and is an excellent way for a company to protect their assets when dealing with a provider they do not fully trust. There are specialized escrow providers.

Virtual machine — A virtual machine is a computer (or other computing component, such as a printer) that does not have physical machinery of its own; instead, its physical machinery is simulated on another machine. Virtual machines have several uses, including individualized desktops that can be accessed from any physical machine, running programs for another operating system and software testing. VMs are a commonly offered cloud service under the title DaaS, desktop as a service. One example is Microsoft's Azure VM.

Transparency

One of the most important concepts in application design, especially cloud computing, is the principal of transparency. This can be a confusing topic to laypeople because the term's definition in the technology world is almost the exact opposite of the word's everyday meaning. Conceptualize software transparency like transparency in a window. A system is transparent if it is invisible to the everyday user, who sees through it to the data behind it. A system is not transparent if you can see it or must understand its technical functioning to use it, as the ordinary definition of the term seems to suggest. Such a system could be understood as "blocking" the user's view of the underlying data. Transparent systems are sometimes also called black boxes because their inner workings are concealed from users, who only see the inputs and outputs.

The mountains are so visible through the left window that you are barely aware the glass is there, but the fog on the right window makes the glass obvious even though you can still see through it. These windows are an analogy of good (left) and bad (right) transparency.

There are many forms of transparency. Some of the most commonly-cited forms are summarized below:

Access transparency — The way a user accesses different resources in a system should look uniform to the user and be primarily out of his awareness.

Distribution/Network/Data organization transparency — How data is stored in a network or distributed database and how such a system operates should not be required knowledge for the user.

Location transparency — The system should function the same from the user's point of view regardless of where the data is stored or where the part of the system handling the user's requests is located.

Migration transparency — The user should not need to be aware of whether the assets he is using can be relocated.

Relocation transparency — The user shouldn't notice whether the assets he is using or the data he is accessing is relocated within the system while he is using it.

Naming transparency — Once a piece of data or asset has been assigned a variable name, it should be reliably accessible through that name, without any additional input.

Replication transparency — Where a distributed database utilizes replication, or a system uses many replicated nodes, a user should not be directly aware that he is accessing one of many copies of the same information.

Concurent transparency — The user should be unaware of how many other users are using the same resources he is.

Persistence transparency — The user should be unaware of whether the data they are accessing is kept in temporary memory or on a (permanent) hard drive.

Fragmentation transparency — The user should not be aware of whether and how a database is fragmented and what fragment(s) he is currently using.

Design transparency — The user should not have to know how the database they are using is designed.

Execution transparency — The user should not have to know on what system and where in the world his transaction runs.

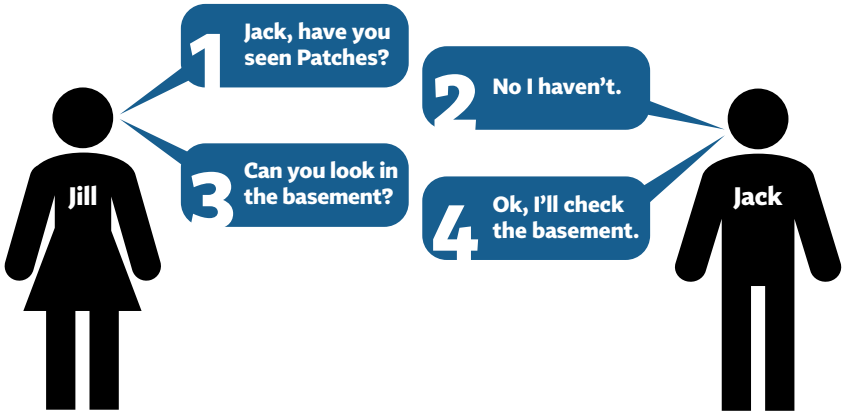
Security transparency — The user should be hassled with security details, including passwords, as little as possible and most of the functioning of the system's security should be outside of the user's conscious awareness.

Beyond promoting general application cleanliness and usability, transparency provides many positive benefits to a system. By keeping implementation details hidden from the end user, it is harder for malevolent actors to find vulnerabilities to exploit. In addition, transparency is an important way that technology companies protect trade secrets and other important IP from reverse-engineering while still allowing users to benefit from them. Even apparently innocuous breaches of transparency can sometimes create an opening for a user to learn crucial system details through repeated and systematic use of the application by the user. Transparency also promotes the reliability and effective maintenance and upgrade of a system by divorcing user commands from actual functioning. It is therefore possible to make sweeping changes to the system's architecture or functioning without disturbing the users. This insulates the system from change fatigue, or user frustration with changes in the functioning of their technology.

Abstraction layers and interfaces: The basic architecture of cloud applications

One of the best ways of achieving transparency is abstraction layers, or different portions of a system conceptualized as layers on top of one another that pass information and translate commands between one another, with commands getting less transparent and more technically complex at lower levels. For this reason, cloud applications make extensive use of them. Abstraction layers are also incredibly important in computing generally because they allow software and hardware components produced by different organizations at different times to function together in the same system. All abstraction layers communicate with each other using interfaces, mutually recognized communication languages and conventions for conveying commands and data.

To illustrate how interfaces work, consult the below (rather stilted) conversation between a couple looking for their cat. In this figure, Jill represents a higher level of abstraction while Jack represents a lower one. The blue speech bubbles represent the interface between them (conversation). In speech bubble one, Jill asks Jack if he has seen the cat, implicitly instructing him to look around a bit and try to remember what he has seen. Jack responds in speech bubble two passing information back to Jill that he hasn't. Jill then responds by asking Jack to check the basement in speech bubble three, who confirms he will in bubble four before leaving. A clear majority of interfaces use a similar method of call-and-response, known as a request-response or request-reply messaging pattern.

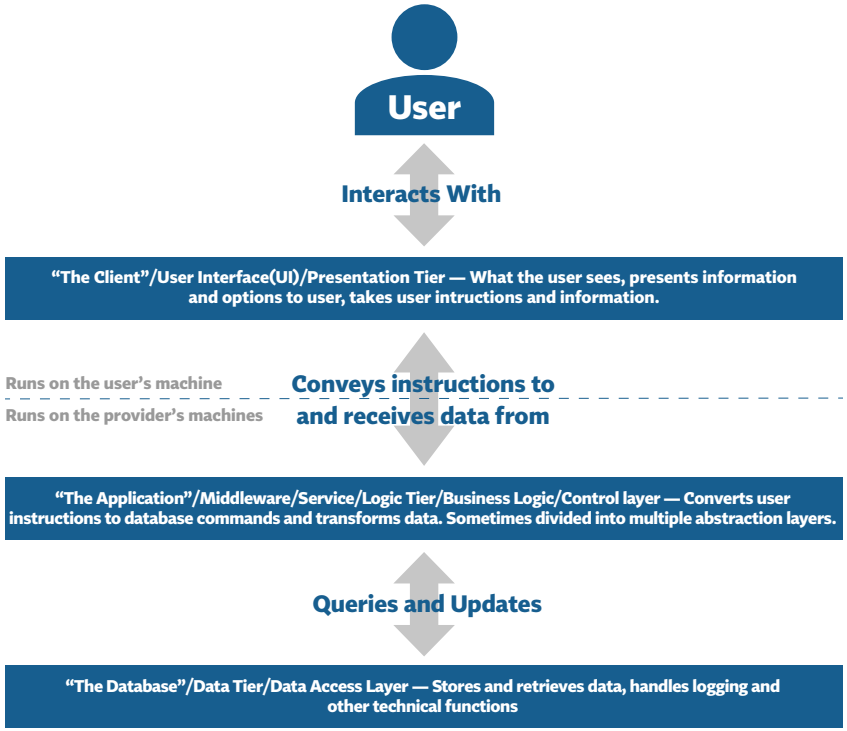


This couple looking for their cat is an example of a request-response messaging pattern, with Jill acting as the higher level of abstraction and Jack the lower.

The names of the four major abstraction layers in individual computers are widely known: software (individual applications the user uses), operating systems or kernels (which translate between software and firmware and manage the computer), firmware (which translate generalized instructions from the operating system and moves the hardware accordingly), and hardware (the physical machinery). The operating system is the lynchpin: the interfaces it provides allow software and manufacturer-produced firmware to interact through it, permitting computers with dramatically different physical designs, and produced by manufacturers on the far side of the world from each other, to run the same programs.

This is the reason why most programs in the past were written for the most popular operating system, Windows. Certain modern programming languages have solved this problem. A notable example is Java, which bills itself as "write once, run anywhere." These languages add an additional layer of abstraction, the interpreter, an operating system-specific program that can construe the application-unspecific code and issue operating system commands. Several popular video game developers, notably Mojang (Minecraft), have leveraged Java and other WORA languages to make their games available to a wide diversity of operating systems without having to adapt their code to each individual operating system. This is one example of the power of abstraction layers to solve compatibility issues and other computing problems. Web browsers, like Google Chrome, work very similarly when they interpret WORA languages, like JavaScript, embedded in webpages.

Abstraction layers do not have to be located on only one machine or directly involve physical hardware. Indeed, every website uses at least two abstraction layers: the client, a local instance of the website that loads and runs on the user's machine, and the server, which provides and updates websites for the client and handles data processing on the client's request. This is called a client-server architecture. The relationship between the client and the server or group of servers is a service. Many cloud applications expand on this with a multitier architecture, also called an n-tier or multilayered architecture, where each abstraction layer (or tier) used in the application is located on a different server or group of servers and serves a different function. The three-tier architecture is the most common multitier architecture:



A diagram of the three-tier architecture. The user interacts with the client, which passes information into the application, which then passes commands to, and receives information from, the database.

It is possible to allow users to create their own custom programs using the host's system on levels of abstraction above the client. This is accomplished by developing an application programming interface for the client. An API is a special interface provided by the developers of an application that will allow other developers to incorporate it into their own applications. It is like a cyber trailer hitch. An enormous number of popular web applications have publicly available APIs, including Twitter, Instagram, Gmail, YouTube, Spotify and even The New York Times. APIs can be widely available to the public, behind a paywall or fully restricted to a select group. When adding API restrictions, treat the API as if it were a webpage with the same capabilities. Because every API has the potential for misuse, even internal ones, API proposals should be vetted carefully and in depth by privacy professionals. APIs that involve database querying, even of only one record at a time, are likely worthy of a privacy impact assessment.

An alternative to the centralized client-server architecture is called peer-to-peer architecture, where all the computers in the network are equals and divide computational tasks between one another. Until 2014, Spotify utilized a P2P system in its streaming service and users could receive a requested music file from the cache of another user who had recently listed to it rather than from Spotify itself, saving the company on infrastructure costs and increasing the site's response time and loading speed. As the company's own infrastructure expanded, the need for the P2P system decreased, and it was eventually discontinued.

Test your knowledge

See if you can apply what you've learned to identify whether each of the following descriptions from fictional cloud vendors is "brilliant" (i.e., the description is concise and describes a working product), "bull" (i.e., the description is unnecessarily verbose in a way that is potentially misleading, but describes a working product), or "bunk" (i.e., the product cannot work as described).

1. The architecture diagram provided by ConfreGo, an online conference vendor, shows that their system's service and business logic tiers are separate systems.

2. HappyClient, a customer service chat portal, describes itself as being designed with "perfect system transparency." Elsewhere, the technical description states, "We believe it is important for users to know where their requests are being handled. Therefore, users always know where their version of the chat client is running."

3. Hardhat Hardware says the following about its new Reliant 4k computer line: "Our special no-interface system allows the hardware and operating system to interact directly without any annoying firmware slowing things down."

4. Website design and hosting service Cordz says this about itself: "Our webpage designer SaaS combines with our hosting PaaS and its patented request-response interface to give you the best service possible."

5. Moon Macrosystems says the following about its new product, Bali: "Our Bali FaaS is read once, write anywhere. With it, you can write an application that you can run on any computer."

Answers:

1. Brilliant: The three-fold architecture of most modern online applications is a suggestion, not a hard-and-fast rule. The middle tier is often divided, with separate business logic (the portion of the program that manipulates data according to the requirements of the business) and service (which performs transmissions tasks to the client) tiers being the most frequent division.

2. Bunk: If users know where the version of their chat client is being run, this violates execution transparency and thus the application isn't perfectly transparent. Additionally, it could pose a security risk as an attacker could use this to determine where physical structure of the company's IT infrastructure.

3. Bunk: Interfaces are required to communicate between any two abstraction levels, nothing will work without an interface. There must be an interface somewhere in this system. Additionally, if there is no firmware in a computer, that computer's operating system is the firmware and widely distributed operating systems like Windows won't work on it. While such a design works well for a special-purpose electronic device, like a Gameboy, no general computer worth buying would have such an architecture, it would be expensive for a manufacturer to support, hamper software compatibility, and limit the set of usable removable hardware (keyboards, mice, flash drives, etcetera) to that designed to be compatible with the specific model.

4. Bull: The most glaring issue here is the statement "patented request-response interface." It is very possible that Cordz has patented some specific commands or connection methods in their internal interface, but they cannot patent the actual request-response model, which has been around for a very long time and is a basic of any multitier system. "Webpage designer SaaS" means that the webpage designer is being offered over the internet and "hosting PaaS" could simply be rendered "hosting service." "Our online webpage designer and hosting service uses a patented variation of a common technique to give you the best service possible" is a much more comprehensible but equivalent statement. The statement "best service possible" is ambiguous in this context and could either mean the best customer experience or the most reliable webhosting. Overall, this statement is peppered with technical jargon designed to make Cordz, which offers a very common and increasingly commoditized product, look more technically revolutionary than it really is.

5. Bunk: Bali is a FaaS, "functions as a service," which means that everything needed to write and run it is online and not on any specific individual's computer. Unless Moon puts out a separate interpreter, which they likely do not, there is no independent way to compile the code. While the statement is true on a certain level, any code written in Bali can be run on any computer, it is also untrue on another level: To run any Bali application, a user must be connected to the internet. Therefore, Bali is not truly WORA; a computer without an internet connection cannot run it. This question is very tricky because a FaaS looks and acts just like a programming language but with one major difference: Its code is written and compiled online, not on an individual user's own computer. If you caught this distinction, you have an excellent grasp of the material and what a FaaS truly is.