From Pots and Vats To Programs and Apps

How software learned to package itself

Gordon Haff
William Henry

Red Hat

FROM POTS AND VATS TO PROGRAMS AND APPS

How software learned to package itself

Gordon Haff
William Henry

Gordon Haff
2403 Main Street
Lancaster, MA 01523
bitmason@gmail.com

This book uses Palatino Linotype and Gill Sans MT typefaces (and Interstate for the cover).

ISBN-13: 978-1548927264
ISBN-10: 1548927260
First edition, August 2017

About the authors

Gordon Haff (right) is technology evangelist for Red Hat, the leading provider of commercial open source software. He is a frequent speaker at customer and industry events. He writes for a variety of publications including The Enterprisers Project, opensource.com, Connections, and TechTarget. His Cloudy Chat podcast includes interviews with a wide range of industry experts. He also works on strategy for Red Hat's hybrid cloud portfolio and other emerging technology areas such as IoT, Blockchain, AI, and DevOps.

Prior to Red Hat, as an IT industry analyst, Gordon wrote hundreds of research notes, was frequently quoted in publications such as *The New York Times* on a wide range of IT topics, and advised clients on product and marketing strategies. Earlier in his career, he was responsible for bringing a wide range of computer systems, from minicomputers to large Unix servers, to market while at Data General.

He lives west of Boston, Massachusetts in apple orchard country and is an avid hiker, skier, sea kayaker, and photographer. He can be found on Twitter as @ghaff and by email at gordon@alum.mit.edu. His website and blog are at http://www.bitmasons.com.

Gordon has engineering degrees from MIT and Dartmouth and an MBA from Cornell's Johnson School.

William Henry (left) has been heavily involved in container initiatives at Red Hat. He contributed all the manual pages for the docker project, contributed to Project Atomic, and is a contributing author to a leading containers book. He also works on Red Hat’s DevOps strategy as the strategy lead. Most recently he has been advising on DevSecOps (security in DevOps) and software risk management. William joined Red Hat in the office of the CTO, in 2008, focusing on emerging technologies.

William has over 25 years experience developing distributed applications and systems and service oriented architectures for both government and private industry. William's roles have included engineering, professional services, partner alliances, several management and director roles, and he owned a startup that was acquired by a publicly traded company.

He travels extensively, speaking with customers in various industries about how the latest technology shifts will affect how they do business. He has been a guest speaker and/or expert panelist at Red Hat Summit, LinuxCon, ContainerCon, OMG, JavaOne, TheServerSide, SDI, DevOps Summit and many other industry events. He holds both a B.Sc. and M.Sc. in Computer Science from Dublin City University, Ireland and currently lives in Monument, Colorado, USA with his family.

Acknowledgements

The authors would like to thank our employer, Red Hat, for its support in writing this book. Red Hat also contributes to many of the open source projects that we discuss throughout these pages.

We thank our many colleagues, both within Red Hat and elsewhere, without whose insights and efforts this book would not have been possible. Dan Walsh reviewed and provided feedback on the sections of this book dealing with container packaging. Ross Turk also gave us detailed feedback and suggestions.

Colby Hoke wrote the original container and Kubernetes material on redhat.com that we used in modified form. More broadly, we took advantage of various material written and/or edited by the Red Hat content team including Laura Hamlyn and Bascha Harris.

In addition, we'd like to thank the podcast guests whose interviews we have excerpted for this book: Al Gillen of IDC, Chris Aniszczyk of the Open Container Initiative, Dan Kohn of the Cloud Native Computing Foundation, and Mark Lamourine of Red Hat.

Thanks to James Governor and his RedMonk colleagues whose invitation to speak at Monkigras was the inspiration for this book.

Photographs are credited throughout this book. The photograph on the cover is by As Meskens under CC BY-SA 3.0 license.

Table of Contents

Introduction

Packaging was the theme for the annual MonkiGras conference James Governor organized for early 2017 in London. James encouraged ex-analyst colleague Gordon to go "meta" on the topic. (Analysts love meta and metaphors and historical context.) The result was a presentation titled "A Short History of Packaging: From the Functional to the Experiential."

Light bulb moment.

The overall packaging theme of MonkiGras and the research Gordon did for his talk turned out to be a great hook for the two of us to jointly write this book. (We work together at Red Hat and collaborate on a wide variety of DevOps and container-related activity.)

It immediately became clear that protecting contents, conveying information about contents, communicating legitimacy and trust, and enabling transactions were all attributes common to both how packaging in the physical world has evolved and the hot topics in software packaging today. And there was clear overlap with the container and DevOps strategy work that William was focused on for his "day job."

The meta view of packaging highlights critical tradeoffs. Unpackaged and unbundled components offer ultimate flexibility, control, and customization. Packaging and bundling can simplify and improve usability—but potentially at the cost of constraining choice and future options.

Bundling can also create products that are interesting, useful, and economically viable in a way the fully disaggregated individual components may not be. Think newspapers, financial instruments, and numerous telecommunications services examples.

Open source software, composed of the ultimately malleable bits that can be modified and redistributed, offers near-infinite choice. Yet, many software users and consumers desire a more opinionated, bundled, and yes, packaged experience—trading off choice for convenience.[1]

This last point is a critical tension around open source software and, for lack of a better umbrella term, "the cloud" in the current era. Which makes understanding the role that packaging plays not just important, but a necessity. Ultimately, packaging helps open source create the convenience and the ease of use that users want without giving up on innovation, community-driven development, and user control.

Throughout this book, we've placed parenthetical detail, including technical background, that's not necessary to the overall story flow in sidebars such as this one. Some of the discussions around topics such as containers inevitably touch on moderately technical topics related to how operating systems work and applications are designed. Nonetheless, we've endeavored to make this book as interesting and accessible as possible for a broad audience even if a few sections dive into some weeds for a time.

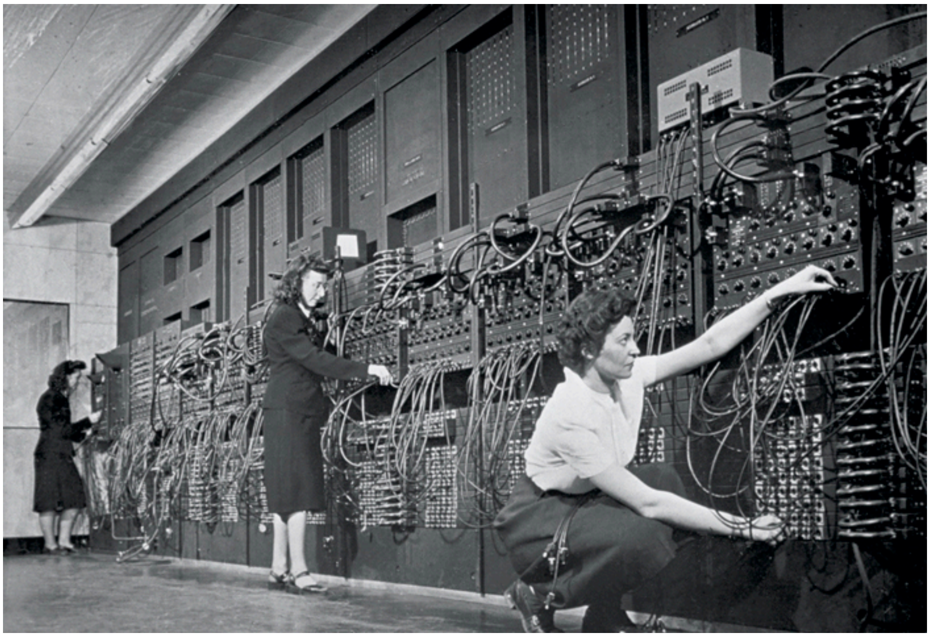
All the information in this book is believed to be correct as of August 2017. However, some of the technology areas covered—containers in particular—are changing rapidly. After all, containers in their current incarnation are only a few years old.

[1] As James' partner at RedMonk, Stephen O'Grady, observed in The Power of Convenience. http://redmonk.com/videos/monki-gras-2017-stephen-ogrady-the-power-of-convenience/

In the Beginning

If we go back far enough, humans didn't package anything. Today, a chimpanzee might use a leaf to collect some water but non-human primates don't store food in any significant way. That's a pretty good indication of the state of affairs in the earliest human hunter-gatherer societies as well.

As a result, most anything stone age humans might have collected had to be consumed both quickly and near to where it was scooped up or gathered. Without some form of packaging, there was no way to carry water or grain to a new location against a future need.



ENIAC, the world's first digital computer at the University of Pennsylvania, had six primary programmers: Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas and Ruth Lichterman. They were initially called "operators." Source: Los Alamos National Laboratory

Our earliest computer programs weren't any more packaged and portable.

ENIAC (Electronic Numerical Integrator And Computer) was the first general-purpose digital computer. Built at the University of Pennsylvania during World War II, ENIAC was programmed by a combination of plugboard wiring and three function tables each of which had 1200 ten-way switches which were used for entering tables of numbers.[2]

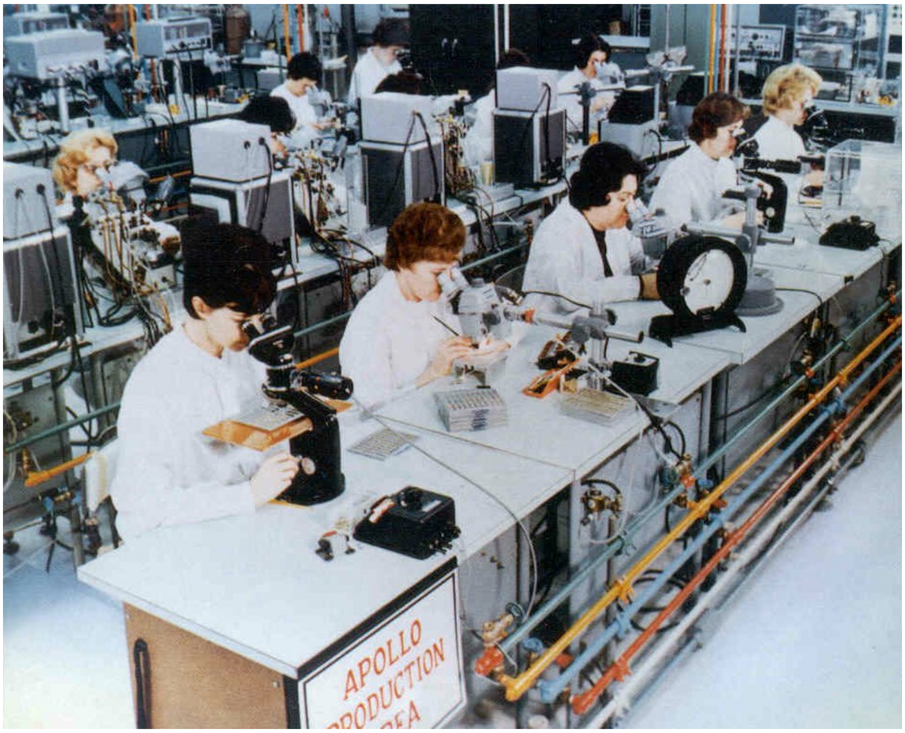
As Franz Alt would write in 1972: "It was similar to the plugboards of small punched-card machines, but here we had about 40 plugboards, each several feet in size. A number of wires had to be plugged for each single instruction of a problem, thousands of them each time a problem was to begin a run; and this took several days to do and many more days to check out."

Unpackaged code would remain around in various forms for a perhaps surprisingly long time. Richard Battin, who led the design of the guidance, navigation, and control systems for the Apollo flights while at the MIT Instrumentation Lab (now named after its founder Charles Stark "Doc" Draper), once recalled a story about the core rope memory used in the Apollo Guidance Computer.

Core rope is a form of read-only memory for computers; the ferrite cores which stored the electrical signals were "woven" to compose programs by a team of ex-textile workers and watchmakers working for Raytheon. It was sometimes nicknamed "Little Old Lady" memory as a result.[3]

[2] http://www.columbia.edu/cu/computinghistory/eniac.html

[3] http://news.bbc.co.uk/2/hi/technology/8148730.stm



"Sewing" rope core memory for Apollo. Source: Raytheon, from the files of Jack Poundstone.

One day, the astronauts toured the facility. As Battin told it, one of the goals was to impress upon the production workers that it was *really* important not to make a mistake in their "sewing" lest these "nice young boys" die.

Programs such as these were one-off affairs, rooted in a single system with no existence outside of that instance of hardware.

Containing

It's hard to say when the first primitive packaging put in an appearance. It probably consisted of leaves, woven grasses (primitive baskets), and other readily available materials such as animal skins. Little evidence has been preserved of these soft and perishable containers.

The oldest examples of pottery yet discovered are remains found in the Xianrendong Cave in the Jiangxi Province in China; they go back about 20,000 years, predating farming and what we generally consider to be civilization.[6] Pottery spread widely in subsequent millennia and fragments are ubiquitous at archeological sites around the world. Such pottery vessels would have been used for storing, cooking, and serving food—as well as carrying water.

The first wine was probably fermented in a pottery container, possibly dating to early Middle Eastern civilizations about 7,000 years ago.[7] Hold that thought for now though; we'll return to packaging for preservation in due course.

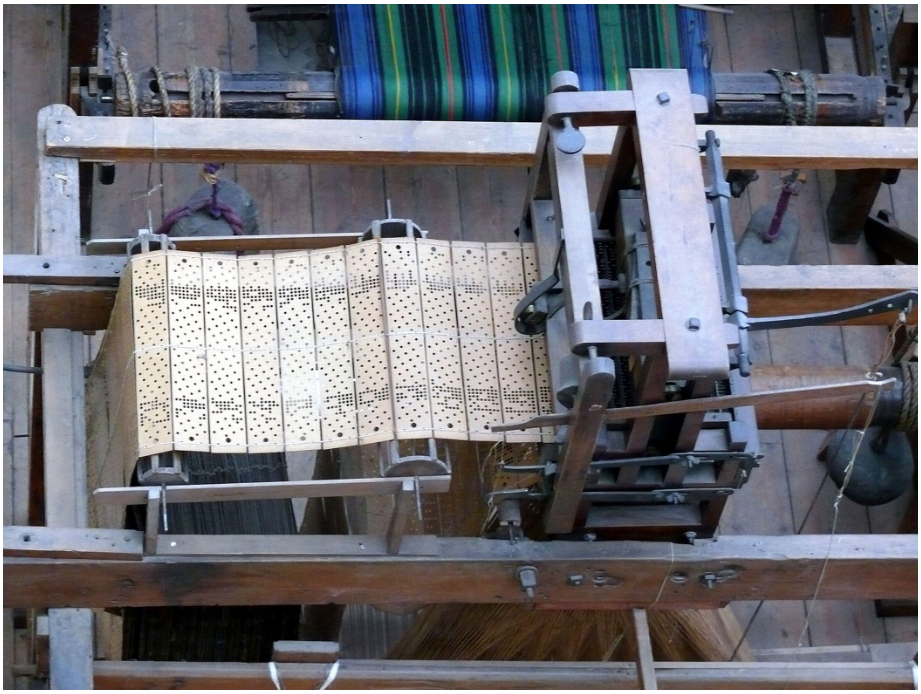
Indeed, potsherds—fragments of pottery—are widely used by archaeologists to date and otherwise better understand when a particular site was occupied and by whom. Characteristics such as temper, form, and glaze help determine the time period and the technologies that were in use at a given site.

In the case of computers, containing instructions and data originally took its cue from earlier forms of storing repeated patterns.

[6] http://science.sciencemag.org/content/336/6089/1696
[7] http://archive.archaeology.org/9609/newsbriefs/wine.html

The precursors to software storage

Perhaps the very oldest such storage can be found in the barrel organ which "owes its name to the cylinder on which the tunes are pricked out with pins and staples of various lengths, set at definite intervals according to the scheme required by the music."[8] The concept dates to the Netherlands in the 15th century but detailed diagrams of a large stationary barrel-organ worked by hydraulic power were first published in 1615 by Jehan van Steenken, a Belgian organ-maker.



Jacquard loom in the National Museum of Scotland, Edinburgh. Source: Ad Meskens / Wikimedia Commons.

[8] https://en.wikisource.org/wiki/1911_Encyclop%C3%A6dia_Britannica/Barrel-organ

The most widely cited precursor to today's data storage came by way of the silk industry in Lyon France in 1725. It was there that Basile Bouchon, a textile worker and son of an organ maker, had the idea to extend the concept of the rotating pegged cylinders used in automated organs to "program" textile weaving. His innovation came from realizing that, before fabricating the expensive metal cylinders used by devices such as barrel organs, the information content had to first be laid out in paper form.[9] For textile weaving, instructions could just be encoded on paper without subsequently creating a costly metal version.

Neither Bouchon's device, nor follow-on refinements by Jean-Baptiste Falcon and Jacques Vaucanson were very successful or effective. But the Jacquard loom, invented by Joseph Marie Jacquard in 1804, was. It substituted a chain of paper cards, each representing a row of the design, for the paper tape and is widely considered to be one of the most important inventions in the history of textiles.

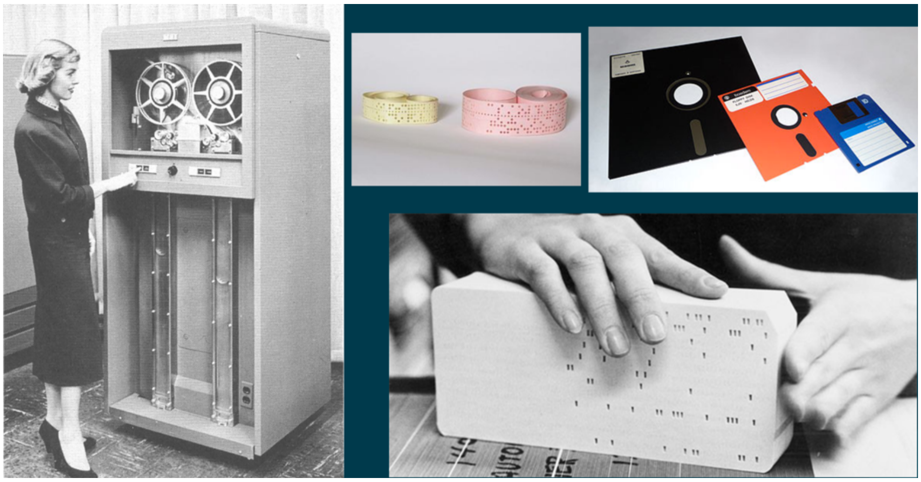
Variegated packaging of data

The punched cards used in automated weaving are a direct ancestor of the punched cards used throughout much of the history of computers. Charles Babbage planned to use them in his never-completed Analytical Engine in the mid-1800s. But they were first actually used in something like computing machinery when Herman Hollerith created a punched card tabulating machine to input data for the 1890 U.S. Census. Hollerith's company would combine with three other firms to become IBM, whose 80-column punched cards were the ubiquitous way to store data until the 1950s (when magnetic data storage started to become common) and remained commonplace for data entry for a couple decades after that.

[9] http://history-computer.com/Dreamers/Bouchon.html

Punched tape had its own parallel history, most associated with teletypewriters and various types of specialized computers such as newspaper typesetting equipment and computer-controlled manufacturing systems. The mechanisms required to write and read a continuous spool of up to one-inch wide paper tape were smaller and simpler than card keypunch machines and card readers—and thus a better fit for equipment that was typically much lower cost and much smaller than that associated with mainframe computing.

The first magnetic media dates to the UNISERVO reel-to-reel tape drive, which was the primary input/output device on the UNIVAC I, the first commercially-sold computer. It recorded on a thin metal strip of half-inch wide nickel-plated phosphor bronze. Shortly thereafter, IBM introduced ferrous-oxide coated tape similar to that used in audio recording. This general type of reel-to-reel drive and media was standard on large computer systems until about the 1990s.



Clockwise from left: Magnetic tape, paper tape, diskettes, and punch cards. Sources: Punched tape and diskettes, Wikimedia. Punched card and tape drive, IBM.

Smaller, cheaper, and more numerous computers sparked a demand for smaller removable magnetic storage. (Reel-to-reel drives were large, complex, and expensive.) In 1972, 3M introduced the Quarter inch cartridge tape (abbreviated QIC, commonly pronounced "quick"), variants of which are still (rarely) in use today. The media is an enclosed package of aluminum and plastic which holds two tape reels driven by a single belt in direct contact with the tape.

Over time, other cartridge tape formats included IBM's 3480 and 7380 families, Digital Linear Tape (DLT) from Digital Equipment Corporation, Linear Tape-Open (LTO), and DDS/DAT. Cartridge tape remains fairly common for large-scale data backup; it's often used in conjunction with large robotic tape library systems, although it's being replaced in that role by high-capacity magnetic disk drives. Today, disk drives that are optimized for capacity rather than per performance are often used for backup and powered-off when not in use to reduce operational costs.

Floppy disk drives are most associated with the PC era but the original 8-inch floppy was developed in 1967 at IBM's San Jose, California storage development center. It was designed as a reliable and inexpensive system for loading microcode (essentially the initialization system) into their System/370 mainframes.

Shugart Associates subsequently developed the 5-¼-inch format diskette for a desktop word processing system that Wang Laboratories was developing in the late 1970s. This form-factor was widely-used in many of the early PCs including the Apple II and the original PC. One or two floppy drives often served as the only persistent storage in these machines although, once hard disk drives dropped in price, "floppies" were increasingly relegated to loading software and backing up data.

Paper tape spools being used for newspaper typesetting, circa 1976. Source: Gordon Haff.

In 1982, the Microfloppy Industry Committee, a consortium of 23 companies, finally agreed upon a 3½-inch media specification after years of competing formats saw spotty use. (It was not actually "floppy" because it used a hard shell.)

The floppy wasn't widely replaced until the adoption of the compact disk (CD). This digital optical disc data storage format, released in 1982 and co-developed by Philips and Sony. was originally developed for audio but became the dominant data transfer and backup medium until a combination of cheap hard drives, high-bandwidth networks, and multi-gigabyte flash memory sticks made it largely redundant.

A higher capacity optical format, Blu-Ray, enjoyed a period of popularity for distributing high-resolution movies for home viewing. However, because of an early-on format battle with HD-DVD and initially expensive writeable media, by the time Blu-Ray might have been broadly interesting as a computer data storage format, it was no longer needed.

What's common to all these formats that have dotted the computing landscape over the years is that they were a way to contain information in a digital form. As with retail shelving and physical packaging, there were attempts to introduce some degree of standardization. But standards are always in something of a war with the desire to differentiate or to optimize for a particular use.

Over time, various innovations to use data storage more efficiently were also developed. For example, especially for uses where storage performance was less important, compression allowed more data to be stored on a given piece of media.

However, as with other forms of packaging, data storage didn't originally exist primarily to make buying or selling software easier—other than incidentally.

Transact

As goods increasingly flowed over long distances and trade became a central part of many economies, traders naturally wanted to streamline both transporting goods and selling them. New designs of pottery containers lent themselves to efficient shipment. One such container was a twin-handled amphora with a characteristic pointed base and elongated shape, which facilitated the transport of oil or wine by ship. The amphorae were packed upright or on their sides in as many as five staggered layers. (You can see an example on the cover of this book.)

Standardization

Amphorae originally differed considerably in shape and size. However, during the Roman empire, the weights and measures used in commerce became more formal. For example, a standard model of an amphora was kept at the temple of Jupiter in Rome; it was called amphora Capitolina. The capacity of this vessel corresponded to the principal Roman measure of capacity for fluids, amphora quadrantal—or just amphora. The measurement derives from the capacity occupied by 80 pounds of wine, about 10 gallons or 39 liters. By law, the quadrantal was connected to the measures of length as its volume was a cubic foot.[10]

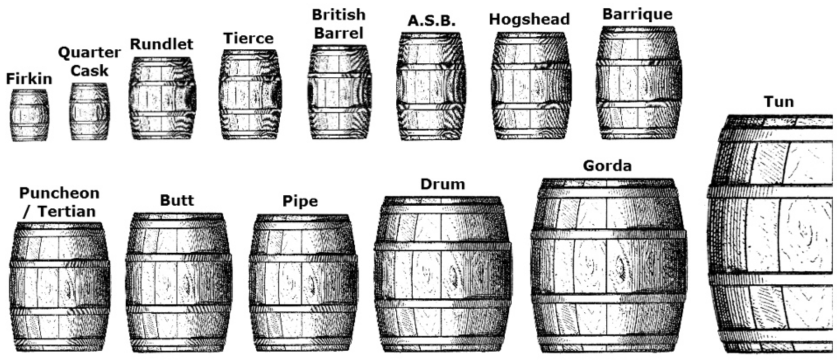
Standardization enables more formalized transactions. An amphora quadrantal might not have signaled anything about the quality of the wine or olive oil it contained. But it at least communicated a predictable quantity.

The Romans also used barrels. But barrels in the form we think of today, made of wooden staves bound by wooden or metal hoops, were more typical further north in Europe—especially in the

[10] William Smith, A Dictionary of Greek and Roman Antiquities, 1875

territories of the Gauls and Celts. Until the twentieth century and the introduction of pallet-based packaging systems, barrels were often the most convenient packaging for shipping all sorts of bulk goods, from nails to whiskey. Bags and crates were also common because they were cheaper, but they were not as sturdy, didn’t protect their contents as well, and could be more difficult to handle.

Barrels of various sizes became standard measures of volume across a broad swath of industries. Firkin, hogshead, gorda, tun, butt, and barrique measures all derive from cask sizes. The practice carried over when steel drums, including the standard 55-gallon steel drum, replaced barrels for many applications. The 42-gallon standard oil barrel volume measurement is still used today throughout the petroleum industry, even though actual physical barrels are no longer used to transport oil.



Some of the historical sizes of barrels (casks).

The gallon (galun or galon in Norman) probably dates to about the time of William the Conqueror, who invaded England in 1066, although the details get fuzzy prior to the year 1300 or so. The liquid version of the gallon was the basis of a system for wine and beer measurements in England. A variety of gallon variants were used in Britain and its colonies at different times and for different purposes. In the early 19th century, the US standardized on the wine gallon, the volume of which was first legally defined during the reign of Queen Anne in 1706. However, in 1824, Britain standardized its gallon by adopting a close approximation to a different gallon variant, the ale gallon or imperial gallon, which is about 20 percent bigger than the US version (4.5 vs. 3.8 liters). Because pints are one-eighth of a gallon in both systems, this is the historical oddity that gives you four extra ounces of beer when you order a pint in a London pub compared to a Boston one.

Another application of barrel-like containers such as kegs.

The shrink-wrapped software era

We see analogs to amphoras and barrels in the way that software packaging can bring together bits so that they can be sold to and consumed by a customer in a standardized way. The shrink-wrapped software era was made possible by the fact that programs could be written onto standard media from which they could be then loaded onto a customer's computer. There are earlier examples of software being delivered on magnetic tape to business users, but selling software in volume to individual consumers brought an even greater need to simplify the delivery of software from the manufacturer to the retailer and from the retailer to the end-user.

It's difficult to identify the first company to sell software that wasn't also hawking hardware (which is to say, the first Independent Software Vendor (ISV)). However, Cincom Systems—founded in 1968—is a good candidate. It sold what appears to be the first commercial database management system not to be developed by a system maker like IBM. Fun fact: Not only is Cincom still extant as a private company in 2017 but one of its founders, Thomas Nies, is the CEO.

Over time, pure-play or mostly pure-play software companies packaging up bits and selling them became the dominant way in which customers acquired most of their software. ISVs like Microsoft selling closed-source proprietary software even became major suppliers of the operating systems and other "platform" software that historically were supplied by vendors as part of a bundle with their hardware.

Linux distributions

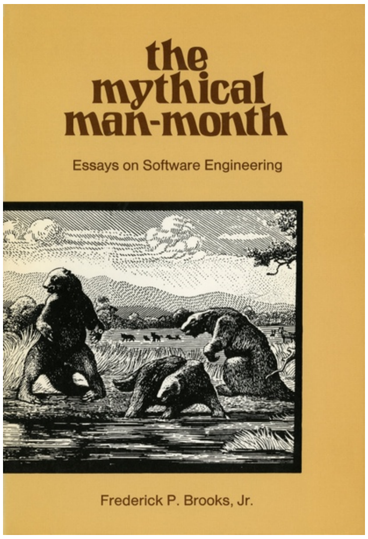
In the world of open source software, distributions brought together the core operating system components, including the kernel, and combined them with the other pieces, such as the utilities, programming tools, and web servers needed to create a working environment suitable for running applications. Although it wasn't the first Linux distribution, Slackware, released by Patrick Volkerding in 1993, was the first that can reasonably be considered well-known. Over the next decade, the number of distributions exploded although only a handful were ever sold commercially. In a 2003 analyst report, Gordon wrote that in addition to the major commercial distributions from Red Hat and SUSE:

> There are a lot of Linux distros out there, ranging from the whimsical to the serious, from the general-purpose to those that are specialists in some function such as real-time computing or for some geographic region such as Asia-Pacific. There's Debian, Slackware, Conectiva, Lindows, Mandrake, SCO/Caldera, Red Flag Linux, and Turbolinux, to say nothing of the literally hundreds of other special-purpose Linux distributions including Bootable Business Card (designed to be booted from a business-card type CD), ChainSaw Linux (for video editing), Xbox Linux (to turn a Microsoft Xbox game console into a Linux computer), UltraPenguin (for SPARC and UltraSPARC), YellowDog Linux (for PowerPC), spyLinux (fits on a single floppy), and the initially alarming and recursively acronymic JAILBAIT.

Distributions were a recognition that an operating system kernel and even the kernel plus a core set of utilities (such as those that are part of GNU in the case of Linux) aren't that useful by themselves.

Commercial open source subscriptions, such as Red Hat Enterprise Linux, further extend the idea of distributions by incorporating support, hardware and software certifications, legal protections, and other things that customers value. This is the next step to creating a more complete experience for buyers through packaging. It's also part of an overall trend to streamline the path from developer to the user. What analyst Stephen O'Grady calls the "power of convenience." Making it easy for users to meet some business need through software is a central aspect of how packaging and software intersect.

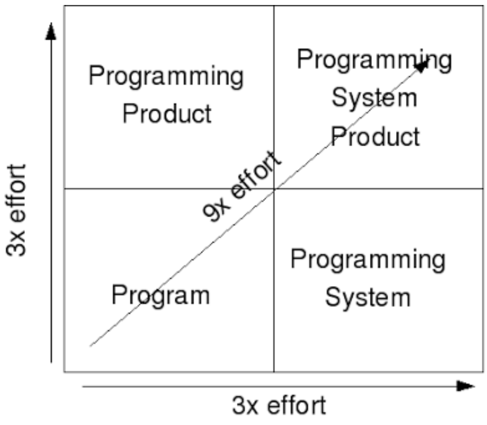
The Product



Fred Brooks is best known for writing *The Mythical Man Month*, a series of essays reflecting on the development of the operating system for IBM's System/360 mainframe which began in the late-1960s. What everyone remembers from that book is the adage that adding more people to a late project makes it even later for reasons of ramp up time, communication overhead, and the inability to divide up many tasks. Nine women can't have a baby in one month and all that. Hence, the book's title.

Programming Systems Products

However, *The Mythical Man Month* kicks off with a different discussion: namely the distinction between a Program and a Programming Systems Product. From Brooks' perspective, evolving the Program into a "truly useful object" required evolving it along two dimensions, as shown in this figure from his book.



In the first dimension the program becomes a programming product, a program that can be run. This involves tasks like testing, documentation, maintenance, and generalization to a range of inputs. In the second dimension, the program becomes a programming system: “a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks.”

Brooks estimated that costs increased by about 3 times along each of these dimensions, resulting in a useful product costing about 9 times the money and effort that went into the original program.

It’s probably worth noting that this discussion is very much flavored by the large system, waterfall development model in which it was rooted. Nonetheless, we see echoes today in humorous aphorisms such as the ninety-ninety rule: “The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” (Attributed to Tom Cargill of Bell Labs.)

Products are a form of packaging.

Products aggregate. This is similar in concept to Brooks’ programming system. In many cases, people prefer to purchase products that include all the parts and dependencies that they need to use a product. There’s a reason that the old Christmas morning “batteries not included” trope was not intended as positive commentary (and has become largely a thing of the past).

Furthermore, finished products often aggregate a *prescriptive* bundle of parts. There are certainly cases where buyers want to exercise maximum control over individual components. But, more commonly, they’re looking for someone else to have done the work of researching and sourcing parts that are to be used together.

Source: The Internet (unknown).

Beyond aggregation

Products generally also go beyond aggregating parts to integrating them. An automobile is not a box of parts. It's a fully integrated assembly that's sold as a complete product. Customers may be offered some options. (The automotive industry is notorious for using option packages to bundle things that many customers want with things that they might not otherwise buy.) However, whatever the specifics, almost no packaged product just throws a bunch of parts in a box. Rather, it constructs and presents a new thing out of an often complicated web of component supply chains.

Brooks' programming product dimension applies even when the nature of the final good means there's "some assembly required." Testing, instructions, support, and (for some types of products) updates are all part of delivering a packaged product to a customer.

Ikea very much sells complete packaged products even if the buyer needs to assemble them. In fact, its packaging is central to both its identity and its business model. For example, the European Logistics Association noted that: "In order to lower logistics costs and increase efficiency in its transportation and warehousing operations, IKEA started an internal competition to reduce unnecessary air in their product packaging. This 'Air hunting competition' focused on removing as much air as possible from packaging and thereby increasing true product volume during transportation and storage."

We see aspects of creating both programming systems and programming products in the open source software world.

Turning open source into products

Entire new categories of software are open source by default, in part because of the success of the community development model. Open source underpins the infrastructure of some of the most sophisticated web-scale companies, like Facebook and Google. Open source stimulates many of the most significant advances in the worlds of cloud, big data, and artificial intelligence. Furthermore, as new computing architectures and approaches rapidly evolve for cloud computing, for big data, and for the Internet of Things (IoT), it's also becoming evident that the open source development model is extremely powerful because of how it allows innovations from multiple sources to be recombined and remixed in powerful ways.

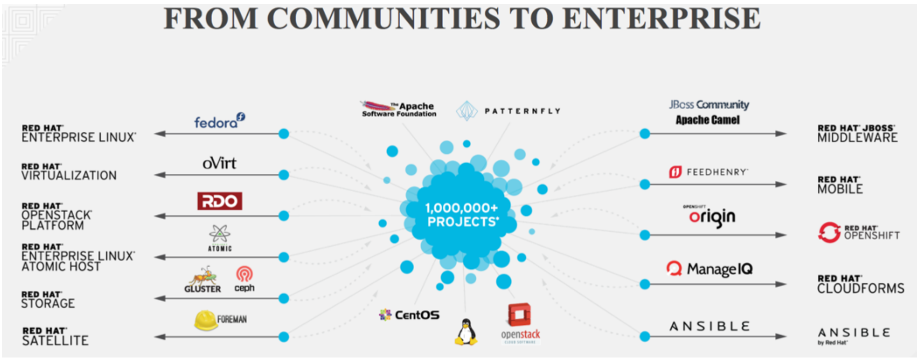
But the huge amount of technological innovation happening around open source can be something of a double-edged sword. On the one hand, it creates enormous possibilities for new types of applications running on dynamic and flexible platforms. At the same time, channeling and packaging the rapid change happening across a plethora of open source projects isn't easy—and can end up being a distraction from the business goals of a company that's merely using open source software to achieve some objective.

In some respects, you can think of many open source projects as programs in Brooks' parlance. They embody a set of capabilities but they're not always fully fleshed out in the ways that let customers depend on them for critical needs.

Commercial open source subscriptions are about creating programming system products. In other words, they make community open source technologies more usable and supportable by enterprise IT. This usually involves working "upstream" to engage with open source communities and influence technology choices in ways that are important to the users of that software. This takes advantage of the strengths of open source development while maintaining technology expertise to provide fast and knowledgeable product support.

Part of this process is also selecting which upstream projects are in a state that's appropriate for a given customer use. For some uses, this means prioritizing stability and maturity. Other uses are a better match for a rapid development and release cycle that provides the latest technology on current hardware platforms.

Al Gillen, who is responsible for open source research at industry analyst firm IDC, noted in a recent interview that: "As we go up the [software] stack, customers still see value associated with commercialization, so a company that will take your project and make it something that is consumable will provide the support. The reason why that's so valuable is that [customers do] not have to have the expertise on staff."



This graphic shows a number of the upstream community projects that map to supported open source subscription offerings.

Gillen's opinion reflects data that IDC has collected over time. For example, in their DevOps Thought Leadership Survey from 2015, they found that "80% prefer vendor supported Open Source enabled solutions."

Not just support

It's worth mentioning at this point that commercial open source often gets pigeonholed as being about "support," which in turn conjures up an image of support staff at call centers waiting for a telephone call or email. That's a part of it of course.

But subscriptions also provide access to knowledge about using products more generally that goes beyond support in the event of a problem. It can include automated access to knowledge repositories, product documentation, and other resources. This sort of self-service access is often faster and easier than opening a support case.

Commercial open source products also typically include updates and upgrades through a defined product life cycle. This is particularly important when security vulnerabilities happen. During the Shellshock and Heartbleed security incidents, for example, Red Hat customers received the knowledge, patches, and applications needed to verify their exposure and successfully remediate potential issues within hours of the bugs being made public. Subscription products can also carry legal protections and certification agreements with other vendors.

It can even include access to the experts who work with upstream communities on a daily basis in order to solve a problem or prioritize a feature on the roadmap.

As Fred Brooks wrote back in 1975, this packaging makes the difference between a program and a system product that's generally useful for business.

Delivery

We've now arrived at a packaged good, perhaps a complex packaged good, which can be sold and used in a supportable way. But we need to deliver it efficiently.

The container ship metaphor

There's a powerful metaphor for this in the physical world—indeed so powerful and useful (if somewhat flawed as metaphors are wont to be), that many tech folks are a bit tired of hearing about it by now.

The shipping container, as described by Marc Levinson in *The Box: How the Shipping Container Made the World Smaller and the World Economy Bigger*, radically changed the economics of shipping the goods we purchase and use every day. Without the shipping container, the globalization of goods would never have happened—at least not at the scale it has.

Container ship MSC Oscar, first visit in Rotterdam. Source: kees torn (MSC OSCAR & SVITZER NARI) CC BY-SA 2.0, via Wikimedia Commons

Containers have been around in various forms since at least the 1800s, beginning with the railroads. In the United States, the container shipping industry's genesis is usually dated to Malcom McLean in 1956. However, for about the next twenty years, many shipping companies used incompatible sizes for both containers and the corner fittings used to lift them. This in turn required multiple variations of equipment to load and unload containers and otherwise made it hard for a complete logistics system to develop.

But around 1970, standard sizes and fittings and reinforcement norms were developed (with all the political jostling between the incumbents that you'd expect). This points to the important role that standards can play. Without the standardization of the shipping container, it would have effectively been just another type of box rather than the component at the heart of an intermodal delivery system.

Existing infrastructure also influences the design of this system.

Individual forty-foot long containers are about the maximum size that can be transported by truck.

The size of container ships is largely constrained by the width and depth of the Panama and Suez Canals. A "Panamax" (or, now, New Panamax or Neopanamax) container ship is the maximum size that can go through the Panama Canal; a "Suezmax" the largest that can go through the Suez Canal. "Malaccamax" ships have the maximum draught that can traverse the Strait of Malacca between the Malay Peninsula and the Indonesian island of Sumatra.

In a totally different context, there's a good argument that the Segway, a much ballyhooed self-balancing "personal transportation vehicle," failed, not so much because of price or poor design, but because it wasn't a good fit with either existing sidewalks or roads (which also inhibits widespread bicycle use in many American cities). Packaging systems are most effective when they fit within existing constraints and infrastructure—or at least can play off them.

As important as standards to the adoption of containers were changes to the labor agreements at major ports. When containers were first introduced, existing labor contracts negated much of their economic benefit by requiring excess dockworkers or otherwise requiring processes that involved more handling than was strictly necessary. By reason of both new labor agreements and infrastructure, containerization allowed the Port Newark-Elizabeth Marine Terminal to largely eclipse the New York and Brooklyn commercial port. Making the best use of packaging systems can require making changes to processes and workflows.

The container embodies a lot of interesting lessons for how technologies evolve more broadly—and how everything old is new again. How does this apply to software packaging?

The rise of software containers

Some of the core technologies underpinning (software) containers are nothing particularly new.

The idea behind what we now call container technology first appeared in 2000 as a way of partitioning a FreeBSD (Unix) system into multiple subsystems, aka "jails." Jails were developed as safe environments that a system administrator could share with multiple users inside or outside of an organization. The intent was that, within a jail, software ran in a modified environment. It had access to most of the usual system services but was walled in so that it couldn't escape and compromise other users and tasks. Jails weren't widely used and methods for escaping the jailed environment were eventually discovered.

Containers were initially viewed as a more lightweight isolation alternative to hardware virtualization. But it's their ability to package applications and their dependencies that has triggered much of the current interest. Source: Illuminata.

In 2001, an implementation of an isolated environment made its way into Linux, by way of Jacques Gélinas' VServer project. As Gélinas put it, this was an effort to run "several general purpose Linux servers on a single box with a high degree of independence and security." Once this foundation was set for multiple controlled userspaces in Linux, pieces began to fall into place to form what is today's Linux container.[11]

[11] Other container implementations included SWsoft's (now Parallels) Virtuozzo and Sun Microsystems' Solaris. The Solaris 10 implementation is probably what most popularized the "containers" term, which was Sun's marketing name for isolating workloads within an operating system. Solaris containers first appeared in a beta release in February 2004. (Sun's technical docs used the "zones" moniker for the same thing.) IBM also introduced containers in AIX which were unique in that they allowed for moving running containers between systems.

Like other types of software partitions (including hardware virtualization), a container presents the appearance of being a separate and independent operating system—a full system, really—to anything that's inside. But, like the workload groups that containers extend, there's only one actual copy of an operating system kernel running on a physical server.

From a technical perspective, containers build off the concept of a process, which is an instance of a computer program containing its program code and its current activity. Although a process is not truly an independent environment, it does provide basic isolation and consistent interfaces. For example, each process has its own identity and security attributes, address space, copies of registers, and independent references to common system resources.

The original BSD Unix jails took advantage of chroot, a Unix/Linux operation that changes the root directory for the current running process. One can see how this benefits Linux containers. While depending on the underlying kernel, a completely different root file system, including the Linux distribution libraries and binaries, can be located at the changed root.

The operating system causes the applications running in each container to believe that they have full, unshared access to their very own copy of that operating system when, in fact, they're sharing the services of a single host operating system. (By contrast, hardware virtualization requires that each partition include an individual copy of a guest operating system.) This also points to why the Linux operating system is so integral to Linux containers; container performance, isolation, and security all depend on inherent operating system capabilities.

Over time, more technologies combined to make this isolation approach a reality. Control groups (cgroups) is a kernel feature that controls and limits resource usage for a process or groups of processes. Systemd, an initialization system that sets up the userspace and manages their processes, is used by cgroups to provide greater control over these isolated processes. These technologies, while adding overall control for Linux generally, were also the framework for how environments could be separated successfully within a single copy of an operating system.

Advancements in user namespaces were the next step. Namespaces isolate and virtualize system resources in a group of processes. They essentially allow changes within one container to be made without affecting other containers on the system.

User namespaces allow per-namespace mappings of user and group IDs. In the context of containers, this means that users and groups can have privileges for certain operations inside a container without the need to give them those same privileges outside the container. For example, an administrator can give someone uid 0 (root[12]) in the container without giving them uid 0 on the underlying system. This is similar to the concept of a jail, but with the added security of further isolation of processes, rather than jails' concept of a modified environment.

The Linux Containers project (LXC) then added some much-needed tools, templates, libraries, and language bindings for these advancements—improving the user experience when using containers. The use of the acronym LXC most often, and correctly, refers to the LXC tools (really tools, templates, and libraries) rather than the idea of Linux containers more broadly.

In the Transact chapter, we discussed operating system distributions. For the purposes of a container discussion, the operating system can be broken down into two areas.

[12] i.e. Essentially complete control.

First, there's the operating system kernel which schedules and manages running programs, or processes, and the resources associated with those processes.

However, the operating system distribution, whether Fedora, Ubuntu, Red Hat Enterprise Linux, or something else, also provides added libraries and applications. For example, almost all Linux distributions include the GNU packages, a widely-used set of utilities and other programs.

For containers to run on a host they only require the host's kernel, often with the addition of modules such as SELinux for additional security, and the LXC tools. An application running in the container may also have dependencies on specific packages from a specific distribution. Those packages must then be made part of the container image. GlibC, the GNU C language library, is an example of a common package dependency in many containers.

Containers: From isolating to packaging

So far we've considered containers as an isolation mechanism. However, containers were largely ignored when they were viewed solely through the lens of partitioning workloads, losing out to hardware virtualization for a variety of reasons. This changed when containers became about packaging.

As we've discussed, a Linux container is a set of processes that are isolated from the rest of the system. By providing an image that also contains an application's dependencies, a container can be made into a packaging construct that is portable and consistent as it moves from development, to testing, and finally to production.

In April 2017, Docker made an announcement that changed the way docker, the Linux container tooling project, would be structured and managed.

The core functionality of what was the docker project has now moved into an open source project called Moby. Moby is both a library, for building containers and some of their dependencies like networking and volume management, and a framework for assembling those components.

As of time of publication, the word "docker" refers to several things:

The project formerly known as "docker" (or "upstream docker") was containerization technology that simplified the creation and use of Linux containers. The core of this technology is now in a project called Moby.[13] It is worth mentioning that the commands `docker build` and `docker run` are not part of Moby. They have been moved out of the core and are part of github.com/docker/ui project.

The company, Docker Inc., open sourced their technology to the Moby community and continue to build on the upstream work and provide supported products called Docker Community Edition and Docker Enterprise Edition.

If this is a bit confusing to the reader it is because not all the details associated with the new upstream project have been fully fleshed out as of the publication of this book. However, the Open Container Initiative (OCI) helps to abstract away the core needs of image and runtime standardization. As a result many organizations have been able to focus on enterprise concerns like container orchestration (e.g. Kubernetes) and security. Much of the tooling has evolved for building and managing container images and running containers. Projects such as Buildah[14] for creating container images, Skopeo[15] for managing image registries, and cri-o[16] for abstracting OCI-compliant runtimes from orchestration engines have taken advantage of OCI standardization and remove the dependency on Docker Inc.'s products.

---

[13] At publication time, the docker command-line interface was not part of the Moby project.
[14] https://github.com/projectatomic/buildah
[15] https://github.com/projectatomic/skopeo
[16] https://github.com/kubernetes-incubator/cri-o

Imagine you're developing an application. You do your work on a laptop and your environment has a specific configuration. Other developers may have slightly different configurations. The application you're developing relies on that configuration and assumes specific files are present. Meanwhile, your business has test and production environments which are standardized and have their own configurations and their own sets of supporting files.

You want to emulate those environments locally as closely as possible, but without the work of recreating the server environments manually. So, how do you make your app work across these environments, pass quality assurance, and get your app deployed without massive headaches, rewriting, and break-fixing?

The answer: Containers. The container that holds your application also holds the necessary configurations (and files) so that you can move it from development, to test, to production—without nasty side effects.

That's a simplified example, but Linux containers can be applied in many different ways to problems where ultimate portability, configurability, and isolation are needed. This is true whether running on-premise, in a public cloud, or a hybrid of the two.

How did the industry move from containers as an approach for isolation to an approach for packaging?

Docker Inc. came onto the scene (by way of dotCloud) with their eponymous container technology, initially released as open source in 2013, which combined the LXC tools with further-improved tools for developers, increasing the user-friendliness of containers.

Its most important innovation was in the area of packaging container images. The docker project's image layering technique helped standardize the way Linux container images are built and shipped. Docker subsequently moved control over the standardization effort for container image formats and the container runtime to the Open Container Initiative (OCI).

The OCI, part of the Linux Foundation, was launched in 2015 “for the express purpose of creating open industry standards around container formats and runtime.” This project is focused on determining and setting specifications. Currently there are two specs: Runtime and Image.

The Runtime Specification sets open standards around a filesystem bundle, the structure of supporting files and artifacts in a container, and how that bundle is unpacked by a compliant runtime. Basically, the spec exists to make sure containers work as intended and that all supporting assets are available and in the correct places. The reference implementation of the runtime specification is runC.[17]

A container runtime automates deploying the application (or combined sets of processes that make up an app) inside this container environment. That is, the container runtime starts and stops the container process with the stipulated storage and network resources it requires.

OCI’s Image Specification defines how container images are created. This creation outputs “an image manifest, a filesystem serialization, and an image configuration.”

Container tools use an image-based deployment model. This makes it easy to share an application, or set of services, together with dependencies across multiple environments.

These specifications work together to define the contents of a container image and those dependencies, environments, arguments,

[17] https://github.com/opencontainers/runc

and so forth necessary for the image to be run properly. As a result of these standardization efforts, the OCI has opened the door for many other tooling efforts that can now depend on stable runtime and Image specs. For example, Red Hat has been involved heavily in container registry and container building projects such as Project Atomic, Skopeo, and Buildah. (Of which, more later.)

One of the interesting dynamics with container standardization today is that it reflects an industry that's more willing to adopt standards in areas where gratuitous differences don't actually differentiate but do hurt adoption.

Chris Aniszczyk is the Executive Director of the OCI and he puts it this way:

> People have learned their lessons, and I think they want to standardize on the thing that will allow the market to grow. Everyone wants containers to be super-successful, run everywhere, build out the business, and then compete on the actual higher levels, sell services and products around that. And not try to fragment the market in a way where people won't adopt containers, because they're scared that it's not ready.[18]

A detour into applications

We've been talking infrastructure. The plumbing. But it doesn't really make sense to talk about containerized infrastructure unless we also at least touch on the application architectures that are going to use those containers.

For a time, it was popular to talk about legacy applications and cloud-native applications using the "pets vs. cattle" metaphor.

[18] http://bitmason.blogspot.com/2017/02/podcast-open-container-initiative-with.html

This metaphor is usually attributed to Bill Baker, then of Microsoft. The idea is that traditional workloads are pets. If a pet gets sick, you take it to the vet and try to nurse it back to health. New-style, cloud-native workloads, on the other hand, are cattle. If the cow gets sick, well, you get a new cow.

Pets and cattle roughly corresponded to the Systems of Record and Systems of Engagement taxonomy proposed by consultant Geoffrey Moore (of *Crossing the Chasm* fame).[19] The former were stateful, big, long-lived, scale-up, and managed/maintained at the individual machine level. The latter were assumed to be stateless, small, transitory, scale-out, and managed at the level of the entire application (with individual instances destroyed and recreated in the event of a problem).

As an initial pass at distinguishing between traditional transactional apps and those designed along more cloud-native lines, the metaphor isn't a bad one. "Ants" may be a better fit than "cattle" in that it captures the idea that individual service instances are not only disposable but they work together cooperatively to perform tasks. In any case, the distinction between long-running mutable instances and short-lived disposable ones is broadly relevant.

That said, both the metaphor and the binary distinction break down if you stare too hard at them. For example, many stateless web-tier applications require persistent data storage in their back-end. Nonetheless, the idea that apps are generally shifting to a more services-oriented modular approach is spot-on.

In their purist form, microservices embody concepts like single-function services built and operated by small ("two pizza")[20] teams, independence from the implementation of other functions, and

[19] https://en.wikipedia.org/wiki/Crossing_the_Chasm
[20] Two pizzas can feed the whole team.

communication only through public interfaces. But, whether or not "microservices" apply in the purest sense (or are even the best approach) in a given situation, they point to a general architecture of modularity, reuse, and optimization at the level of the individual function.

This is a great match for container infrastructure. In fact, microservices plus containers represent a general shift to delivering applications through modular services that can be reused and rewired to perform new tasks.

For example, containerizing services like messaging, mobile app development and support, and integration lets developers build applications, integrate with other systems, orchestrate using rules and processes, and then deploy across hybrid environments. Don't think of this as merely putting middleware into the cloud in its traditional form. Rather, this approach effectively reimagines enterprise application development to enable faster, easier, and less error-prone provisioning and configuration for a more productive developer experience.[21]

One of the key ideas behind microservices is that, instead of large monolithic applications, application design will increasingly use architectures composed of small, single-function, independent services that communicate through network interfaces. This approach is better aligned with agile development practices and reduces the unintended effects associated with making changes in one part of a large monolithic program.

Writing apps for containers

Traditional Linux containers use an initialization system that can manage multiple processes. This means entire applications can run

[21] Red Hat does this with JBoss xPaaS Services for OpenShift (Red Hat's container platform).

as one—effectively just as if they were in a virtual machine or on a "bare metal" physical server. However, modern OCI-compliant Linux container technology encourages breaking down applications into their separate processes and provides the tools to do so. This granular approach has several advantages.

Modularity

The current approach to containerization is focused on the ability to take down a part of an application and to update or repair it—without unnecessarily taking down the whole app. In addition to this microservices-based approach, you can share processes amongst multiple apps in much the same manner as service-oriented architectures more broadly.

Layers and image version control

Each container image file is made up of a series of layers. These layers are combined into a single image. A layer is created when the image changes. Each layer is a set of filesystem changes. Layers do not have configuration metadata such as environment variables or default arguments; those are properties of the image as a whole rather than any particular layer.

Each layer can be isolated into an archive (tar) and each of these archives combined into a single archive along with metadata on the layering. Later these layers can be unarchived onto a layered filesystem like overlayfs or similar.

A variety of projects can be used to build images. Upstream docker itself depends on a Dockerfile and a container runtime daemon to build the various layers of a container image. Buildah from Project Atomic can build a container from scratch and does not require any runtime daemon; it can also use a Dockerfile.

The image layers are reused when building a new container image. This makes the build process fast and has tremendous advantages for organizations applying DevOps practices like continuous integration and deployment (CI/CD). Intermediate changes are shared between images, further improving speed, size, and efficiency. Inherent to layering is version control. Every time there's a new change, you essentially get a built-in change-log.

Rollback

Perhaps the best part about layering is the ability to roll back. Every image has layers. Don't like the current iteration of an image? Roll it back to the previous version. This further supports an agile development approach and helps make CI/CD a reality from a tools perspective.

Rapid deployment

Getting new hardware up, running, provisioned, and available used to take days. And the level of effort and overhead was burdensome. OCI-compliant containers can reduce deployment to seconds. By creating a container for each process, you can quickly share those similar processes with new apps. And, because an operating system doesn't need to restart in order to add or move a container, deployment times are substantially shorter.

Think of technology as being in support of a more granular, controllable, microservices-oriented approach that places greater value on efficiency.

Orchestration

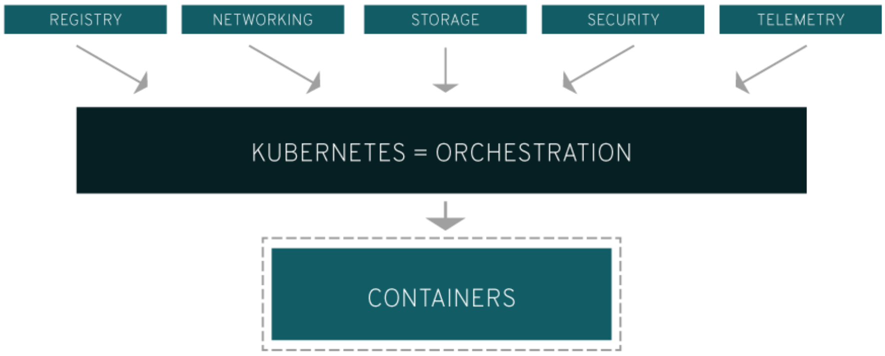
An OCI-compliant container runtime, by itself, is very good at managing single containers. However, when you start using more and more containers and containerized apps, broken down into hundreds of pieces, management and orchestration can get tricky. Eventually, you need to take a step back and group containers to deliver services—such as networking, security, and telemetry—across your containers.

Furthermore, because containers are portable, it's important that the management stack that's associated with them be portable as well.

That's where orchestration technologies, like Kubernetes, come in.

Kubernetes, or k8s (k, 8 characters, s... get it?), or "kube" if you're into brevity, is an open source platform that automates Linux container operations. It eliminates many of the manual processes involved in deploying and scaling containerized applications. In other words, you can cluster together groups of hosts running Linux containers, and Kubernetes helps you easily and efficiently manage those clusters. These clusters can span hosts across public, private, or hybrid clouds. (Although, for performance and other reasons, it's often recommended that individual clusters should be limited to a single physical location.)

Kubernetes was originally developed and designed by Joe Beda, Brendan Burns, and Craig McLuckie of Google. Google had been using a similar platform, Borg, to manage containers internally. The lessons learned from using it became the primary influence behind the Kubernetes technology. The seven spokes in the Kubernetes logo refer to the project's original name, "Project Seven of Nine." Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation (under the Linux Foundation) in 2015.



Kubernetes provides an orchestration layer on top of containers.. Source: Red Hat.

As Dan Kohn, the Executive Director of the Cloud Native Computing Foundation notes:

> It's one of the most exciting software projects on the Internet today. It's also one of the highest velocity projects by almost any metric. Number of commits per day, number of companies participating, number of developers participating, total volume of issues, pull requests. It's probably just second or third behind Linux itself in terms of the velocity that it's been able to keep up.
>
> Even more than that, it's just the fact that it's out there solving real problems for users, for enterprises, for startups, all kinds of companies today, both in the public cloud and bare metal and private clouds. Containerization is this trend that's taking over the world to allow people to run all kinds of different applications in a variety of different environments.
>
> When they do that, they need an orchestration solution in order to keep track of all of those containers and schedule them and

> orchestrate them. Kubernetes is an increasingly popular way to do that.[23]

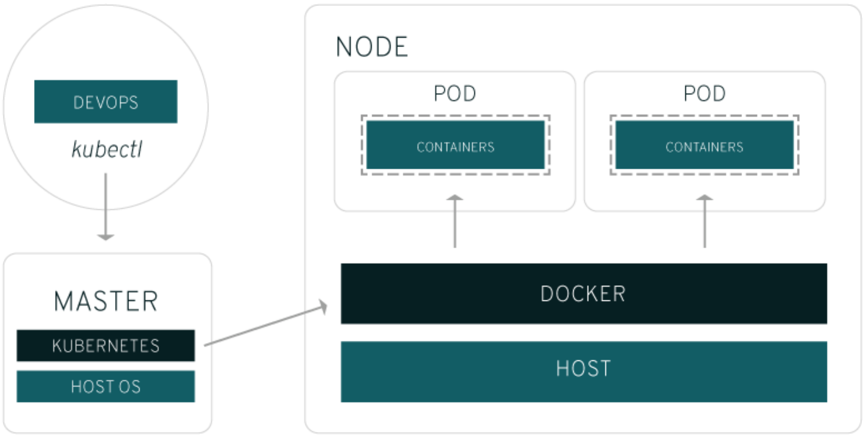
Orchestration allows you to interact with groups of containers at the same time, scheduling and implementing container registry, networking, storage, security, and telemetry services. Once you scale to a production environment and multiple applications, it's clear that you need multiple, co-located containers working together to deliver the individual services. This significantly multiplies the number of containers in your environment and as those containers accumulate, the complexity also grows.

Kubernetes fixes a lot of common problems with container proliferation by structuring containers together into a ″pod.″ Pods add a layer of abstraction to grouped containers, which helps you schedule workloads and provide necessary services—like networking and storage—to those containers. Other parts of Kubernetes help you load balance across these pods and ensure you have the correct number of containers running to support your workloads.

With Kubernetes—and with the help of other open source projects like Atomic Registry, flannel, heapster, OAuth, and SELinux—you can orchestrate all parts of your container infrastructure.

Kubernetes provides a platform to schedule and run containers on clusters of physical or virtual machines. More broadly, it helps you fully implement and rely on a container-based infrastructure in production environments. And to do so in a way that automates many operational tasks.

[23] http://bitmason.blogspot.com/2017/02/podcast-cloud-native-computing.html



Kubernetes orchestrates pods of containers to compose applications. Source: Red Hat.

Because of the standardization of containers through OCI, technologies like Kubernetes can manage containers better and automate critical tasks. At a high level, these include orchestrating, scaling, and maintaining the health of apps and containers running across distributed environments. Kubernetes can also mount and add storage to run apps that require persistent access to a specific set of data. Kubernetes is also attuned to modern service deployment practices—for example, the use of blue-green deployments to introduce and test new features without affecting users.[24]

Furthermore, there is an effort ongoing to take advantage of OCI runtime standards and remove the direct Kubernetes to docker runtime dependency and instead use an abstraction that can use

[24] The blue-green deployment approach does this by ensuring you have two production environments, which are as identical as possible. At any time one of them, let's say blue for the example, is live. As you prepare a new release of your software you do your final stage of testing in the green environment. https://martinfowler.com/bliki/BlueGreenDeployment.html

any underlying OCI-compliant runtime. This effort, called cri-o, means that the kubelet[26] can use any OCI-conformant runtime.

Kubernetes relies on additional projects to provide the services developers and operators might choose to deploy and run cloud-native applications in production. These pieces include:

- A container registry, through projects like Atomic Registry. Consider it the application store.
- Networking, through projects like flannel, calico, or weave. Collaborating containers, or microservices, need networking that also needs to be effectively contained so they can communicate within their namespace with other containers.
- Telemetry, through projects such as heapster, kibana, and elasticsearch. Highly automated systems need logging and good analytics on running applications and their containers.
- Security, through projects like LDAP, SELinux, and OAUTH. Containers and their assets need to be secure and contained.

As we dive into these details, you probably begin to see why it's complex to assemble a platform from scratch using just upstream community projects.

Red Hat OpenShift is a complete container application platform that natively integrates technologies like OCI-compliant containers and Kubernetes and combines them with an enterprise foundation in Red Hat Enterprise Linux. OpenShift integrates the architecture, processes, platforms, and services needed by development and operations teams.

Kubernetes runs on top of an operating system (Red Hat Enterprise Linux Atomic Host, for example) and interacts with pods of

[26] A kubelet is the primary "node agent" that runs on each node.

containers running within the operating system on the nodes (physical systems or virtual machines). The Kubernetes master takes the commands from an administrator (or DevOps team) and relays those instructions to the subservient nodes. This handoff works with a multitude of services to automatically decide which node is best suited for the task. It then allocates resources and assigns the pods in that node to fulfill the requested work.

From an infrastructure perspective, Kubernetes doesn’t change the fundamental mechanisms of container management. But control over containers now happens at a higher level, providing better control without the need to micromanage each individual container or node. Some setup work is necessary, but it’s mostly a matter of assigning a Kubernetes master, defining nodes, and defining pods.

The container runtime technology still does what it's meant to do. When Kubernetes schedules a pod to a node, the kubelet on that node will instruct the container runtime to launch the specified containers. The kubelet then continuously collects the status of those containers and aggregates that information in the master. Container images are pulled from a registry onto that node and containers are started and stopped as normal. The difference is that an automated system asks the container runtime to do those things instead of the admin doing so by hand on all nodes for all containers.

And all this enables not just containerizing applications and services but deploying and managing the entire assembly at scale.

Manufacturing the Delivery Process

Ultimately, the goal is to efficiently and repeatedly deliver standardized and tested product in a repeatable way, a process that transformed manufacturing in the physical world over a period of about 200 years beginning in the late 18th century.

It started with standardization., French General Jean-Baptiste Vaquette de Gribeauval promoted standardized weapons in what became known as the Système Gribeauval after it was issued as a royal order in 1765. Standardized boring allowed cannons to be shorter without sacrificing accuracy and range because of the tighter fit of the shells. It also enabled standardization of the shells.

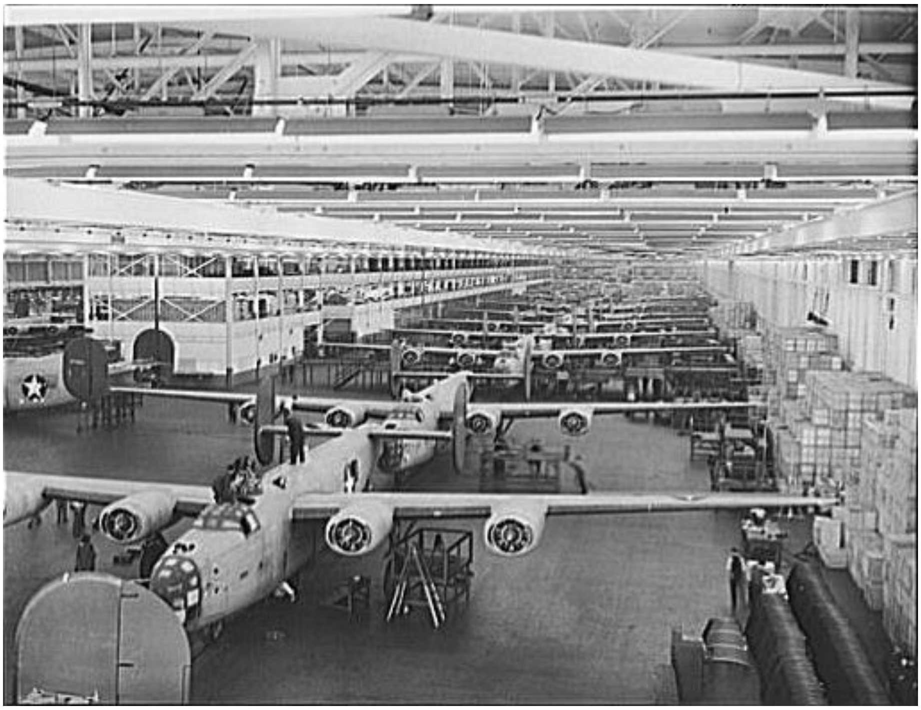
Example of a sailing block. Source: GK Bloemsma, Wikimedia, CC BY-SA.

Gribeauval provided patronage to Honoré Blanc, who attempted to implement the Système Gribeauval at the musket level. By about 1778, Honoré Blanc began producing some of the first firearms with interchangeable flint locks, although these were still carefully made by craftsmen. Blanc demonstrated in front of a committee of scientists that his muskets could be fitted with flint locks picked at random from a pile of parts.

Brunel and Maudsley's sailing blocks brought process to standardization. Marc Brunel, a pioneering engineer, and Maudslay, the founding father of machine tool technology, collaborated on plans to manufacture block-making machinery; the proposal was submitted to the British Admiralty who agreed to commission his services. By 1805, a dockyard had been fully updated with the revolutionary, purpose-built machinery at a time when products were still built individually with different components. A total of 45 machines were required to perform 22 processes on the blocks, which could be made into one of three possible sizes. The machines were almost entirely made of metal thus improving their accuracy and durability. The machines would make markings and indentations on the blocks to ensure alignment throughout the process.

One of the many advantages of this new method was the increase in labor productivity due to the less labor-intensive requirements of managing the machinery. Richard Beamish, assistant to Brunel's engineer son, Isambard Kingdom Brunel, wrote: "...So that ten men, by the aid of this machinery, can accomplish with uniformity, celerity and ease, what formerly required the uncertain labour of one hundred and ten."

It was World War II though that truly brought fully standardized and optimized infrastructure to manufacturing. In *Freedom's Forge,* author Arthur Herman tells the story of how Charles Sorensen of Ford led the construction of the Willow Run manufacturing complex in the early years of World War II. The plant was optimized for the mass production of aircraft, especially the B-24 Liberator heavy bomber. It was the largest manufacturing plant in America because that's what Sorensen's assembly line demanded. He didn't try to squeeze the process into the hangar in San Diego where bomber construction had previously taken place and he introduced processes that resulted in much greater component consistency. At Willow Run, Ford built half of the total B24s, which holds the distinction of being the most produced heavy bomber in history.[28]



B-24 bombers on the Willow Run assembly line.

Finally, the delivery of modern applications using agile development processes, is very much tied to the modern manufacturing thinking that was originally most associated with the Toyota Production System (TPS). Key concepts underpinning this modern approach to manufacturing came from W. Edwards

[28] At least that's the cleaned-up story. In reality, Willow Run had many startup and labor problems and Sorensen was replaced by Mead Bricker in 1943. Consolidated Aircraft also continued to manufacture in San Diego throughout World War II, employing as many as 45,000 workers. Nonetheless, once it got running properly, Willow Run was producing up to 650 B-24s per month and 9,000 total.

Deming, an American who is generally credited with championing the field of statistical process control, building on earlier work by Walter Shewhart. Ironically, Deming was mostly ignored by American manufacturers and ended up being most credited with being an inspiration for what became known as the Japanese post-war economic miracle of 1950 to 1960.

Toyota built on Deming's ideas and incorporated concepts such as lean manufacturing, kaizen (continuous improvement), just-in-time inventory,[29] build-to-order, and systems thinking ("The Toyota Way"). The goal was to make a process as flexible as necessary without stress or "muri" (overburden) since this generates "muda" (waste). It's a long-term philosophy that emphasizes understanding of underlying concepts. However, it also incorporates the idea that tactical improvements can be valuable as well. There's a significant element that's about organization, incentives, and even culture.

We see echoes of all this throughout container platforms like OpenShift and the DevOps approaches used to deliver cloud-native applications using such platforms. Core DevOps principles such as maintaining a single source repository, automating all the things, making builds self-testing, and providing transparency into the code and the process would all be familiar to anyone designing or running a manufacturing system.

At the same time, many of these changes can also be thought of as cultural shifts: craftwork to factories, ad hoc observation to statistical quality control, reduced cycle times, and the empowerment of assembly workers. In essentially all cases, they represent a decisive and deliberate shift from business as usual. We largely agree with JP Morgenthal when he argues that "There is no

[29] It's worth noting that one significant motivation for a system like TPS was inventory reduction—which doesn't really apply to software. Nonetheless, many aspects of the overall philosophy remain highly relevant.

single agreed-upon standard of what culture looks like when DevOps adoption is complete."[30] However, cultural *inputs* like transparency, tolerance of failure, collaboration, leadership, and appropriate incentives are all clearly important.

[30] https://opensource.com/business/15/2/devops-culture-needs-be-created

Preservation

A package can also play a direct role in protecting and preserving its contents. Some of this is essentially inherent to its function. Just the act of containing helps to preserve contents from the elements and containing liquids is the essential first step towards making preservation processes such as fermentation possible.

As Gary Cross and Robert Proctor write in *Packaged Pleasures*:

> Nature is ephemeral—at least that part that grows and dies. When plucked, a plant will spoil or simply disappear… Containerization liberated us from nature, at least a little. This is most obvious with food. Neolithic peoples beginning ten millennia or so ago learned to preserve and pack their nourishment, saving it from decay and also creating thereby entirely new kinds of foods—and sensory delights—in the process. Fermented drink is one notable outcome.

Containerization allowed foods (and drink) to become portable while also being saved for use another day.

Napoleon is often quoted to have said "An army marches on its stomach" (whether or not he actually did). In 1795, the French military offered an award of 12,000 francs (about $50,000 today) to anyone who could devise a practical method for food preservation for armies on the march. A confectioner and chef in Paris, Nicholas Appert, began experimenting with ways to preserve foodstuffs, including soups, vegetables, juices, dairy products, jellies, jams, and syrups. He placed the food in glass jars, sealed them with cork and sealing wax, and placed them in boiling water—a process which, the method of sealing the container aside, would seem familiar to anyone making jam at home today.[31] Appert won the prize, patented his invention, and established a business to preserve a variety of food in sealed bottles.

The history of cans is a bit more convoluted.

Another Frenchman, Philippe de Girard, reputedly demonstrated canned foods at the Royal Society in London in 1810 a few years after Appert's invention. The story is a bit murky[32] but it seems that Englishman Peter Durand took out a patent for this preservation process which could use tinplate cans, among other containers. Solder was used for sealing the can seams.[33]

In 1812, Durand sold his patent to two Englishmen, Bryan Donkin and John Hall, who refined the process and product, and set up the world's first commercial canning factory on Southwark Park Road, London. By 1813 they were producing their first tin canned goods for the Royal Navy.

However, although Girard is often credited with inventing the tin can, some form of tinned iron cylinders appears to have been used by the Dutch navy as early as the mid-1700s. Records show that from 1772 to 1777, while quelling a revolt in what was then Dutch Guiana in South America, the navy was supplied with roast beef packaged in this way. Before the end of the eighteenth century, the Netherlands had a small industry that preserved salmon by canning.[34]

The first can openers weren't patented until 1855 in England and 1858 in the United States. This must have made for an interesting 40 years or so given the instructions like the "Cut round the top near

[31] Lance Day, Ian McNeil, ed. (1996). Biographical Dictionary of the History of Technology.
[32] http://www.bbc.com/news/magazine-21689069
[33] http://www.canmaker.com/online/frequently-asked-questions/
[34] Food Packaging: Principles and Practice, Third Edition, Gordon L. Robertson

the outer edge with a chisel and hammer" to open a can that have been passed down to us.

The reality is that early cans were specialized; the can itself could weigh more than the enclosed food. It wasn't until near the beginning of the twentieth century that food in cans became a common consumer item. The American Can Company was founded in 1901 and was soon producing 90 percent of the tin cans used in the United States.

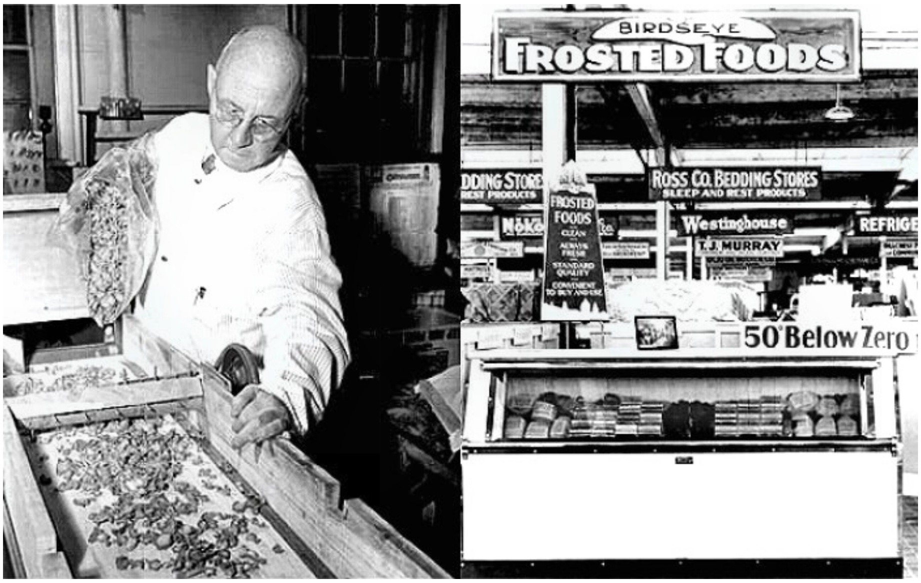
Reducing the weight, bulk, cost, and (most recently) environmental impact of protective packaging has long been an ongoing theme. There's also been a widespread recognition that packaging existing primarily to solve some problem for a manufacturer or retailer, such as reducing theft, shouldn't get in the way of the consumer's experience. Blister packs made of thermoformed plastic are one particularly notorious example.

Online retailer Amazon even offers "frustration free packaging" as an alternative for a wide range of products. It's a good bet that if someone markets an alternative to your product as frustration free, you're probably doing something wrong.

Preservation and the supply chain

Preservation can also intersect with the supply chain through which a product is delivered and the manner in which a product is consumed. Frozen food is a case in point.

Clarence Birdseye is generally considered to be the founder of the modern frozen food industry. In 1925, after a couple of false starts, he moved his General Seafood Corporation to Gloucester, Massachusetts. It was there that he used his newest invention, the double belt freezer, to freeze fish quickly using a pair of brine-cooled stainless steel belts. This and other Birdseye innovations centered on the idea that flash freezing meant that only small ice crystals could form and, therefore, cell membranes were not damaged.



Clarence Birdseye is considered to be the founder of the modern frozen food industry.

A couple of points are worth highlighting. The first is that frozen food depends on a reliable supply chain between the original source of the food and the consumer that can maintain the right temperature for the package. The second is that, in the course of preserving it, food can also be processed in ways that make consuming it more convenient. (For better or worse. Frozen vegetables are easier to be positive about than TV dinners.)

Nor is the task of food preservation complete when a truck leaves the factory loading dock. Packaging and supply chains need to reliably protect, secure, and preserve the overall integrity of contents until they're in a consumer's hands and even beyond.

Securing the software supply chain

In the software world, the packaging of applications and services can likewise protect and secure their contents throughout their life cycle.

Historically, security was often approached as a centralized function. An organization might have established a single source of truth for user, machine, and service identities across an entire environment. These identities described the information users were authorized to access and the actions they were allowed to perform.

Today, the situation is often more complicated. It's still important to have access control policies that govern user identities, delegating authority as appropriate and establishing trusted relationships with other identity stores as needed. However, components of distributed applications may be subject to multiple authorization systems and access control lists.

Insight into and control over complex hybrid environments is a necessity.

For example, real-time monitoring and enforcement of policies can not only address performance and reliability issues before the problems become serious, but they can also detect and mitigate potential compliance issues. Automation reduces the amount of sysadmin work that is required. However, it's also a way to document processes and reduce error-prone manual procedures. Human error is consistently cited as a major cause of security breaches and outages.

Operational monitoring and remediation needs to continue throughout the life cycle of a system. It starts with provisioning. As with other aspects of ongoing system management, maintaining complete reporting, auditing, and change history is a must.

The need for security policies and plans doesn't end when an application is retired. Data associated with the application may need to be retained for a period or personally identifiable information (PII) may need to be scrubbed depending upon applicable regulations and policies.

With traditional long-lived application instances, maintaining a secure infrastructure also meant analyzing and automatically correcting configuration drift to enforce the desired host end-state. This can still be an important requirement. However, with the increased role that large numbers of short-lived "immutable"[35] instances play in cloud-native environments, it's equally important to build in security in the first place. For example, you may establish and enforce rule-based policies around the services in the layers of a containerized software stack.

Taking a risk management approach to security goes beyond putting an effective set of technologies in place. It also requires considering the software supply chain and having a process in place to address issues quickly.

For example, it's important to validate that software components come from a trusted source. Containers provide a case in point. Their very simplicity can turn into a headache if IT doesn't ensure that all software running in a container comes from trusted sources and meets required standards of security and supportability.

It's much like a large and busy port with thousands of containers arriving each day. How does a port authority manage the risk of allowing a malicious or illegal container into the port? By looking at which ship it arrived in and its manifest, by using sniffer dogs and

[35] With lightweight services, the general model is to shutdown and restart instances that have a problem or need to be updated rather than changing the running instance as was historically the usual approach.

other detection equipment, and even by physically opening and scanning the contents.

The verification of shipping container contents is a serious public policy concern because many inspection processes are largely manual and don't scale well. Fortunately, verifying the contents of software containers and packages is more amenable to automation and other software-based approaches.

Most of the vulnerable images in public repositories aren't malicious; nobody put the vulnerable software there on purpose. Someone just created the image in the past but after it was added to the registry, new security vulnerabilities were found. However, unless someone is paying attention and can update those images, the only possible outcome is a registry that contains a large number of vulnerable images. If you just pull a container from one of these registries and place it into production, you may unwittingly be introducing insecure software into your environment.

Many software vendors help secure the supply chain by digitally signing all released packages and distributing them through secure channels. Red Hat also provides vulnerability and errata information in machine readable form so that it can be consumed and acted on at scale — such as through the use of a Security Content Automation Protocol (SCAP) scanner.[36] With respect to containers specifically, the Red Hat Container Registry lets you know that components come from a trusted source, platform packages have not been tampered with, the container image is free of known vulnerabilities in the platform components or layers, and the complete stack is commercially supported.

[36] Red Hat also has partnerships with third parties who have written scanning tools and maintain knowledge bases of vulnerabilities.

Incident response goes well beyond patching code. However, a software deployment platform and process with integrated testing is still an important part of quickly fixing problems (as well as reducing the amount of buggy code that gets pushed into production). A CI/CD pipeline that is part of an iterative, automated DevOps software delivery process means that modular code elements can be systematically tested and released in a timely fashion. Furthermore, explicitly folding security processes into the software deployment workflow makes security an ongoing part of software development—rather than just a gatekeeper blocking the path to production.

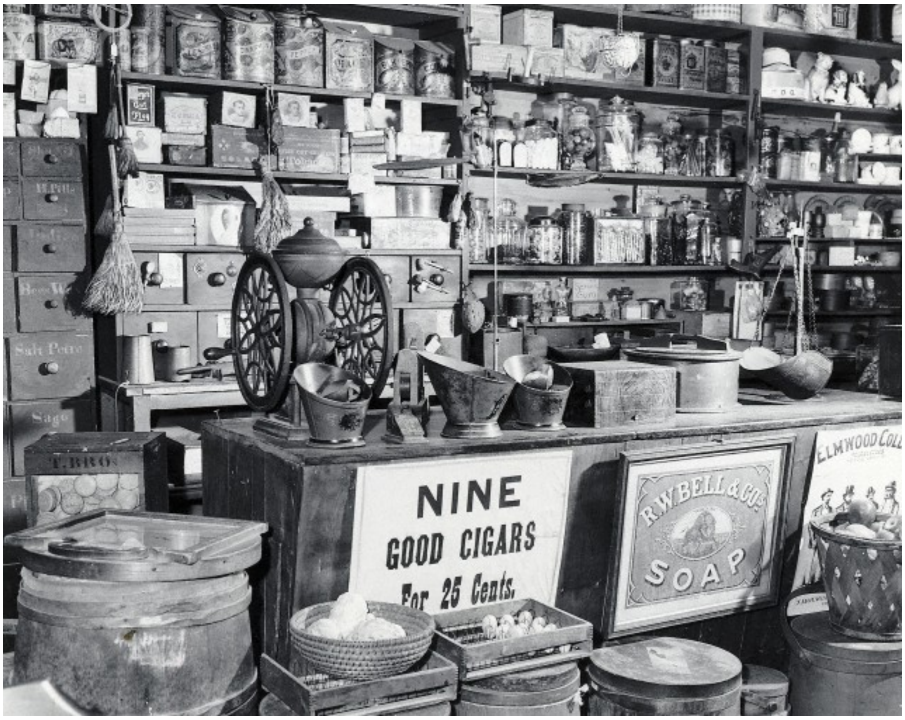
The first part of this book has primarily been about aspects of packaging physical goods and software that are primarily functional. How do we use packaging to sell and deliver a useful product? How do we protect that product? These are table stakes really—the minimum needed to put a product in the hands of a customer.

With this as a starting point, we now move into the realm of the experiential. There was less to be said about software here until recently. This is partly because, for much of its history, computer software was a utilitarian business tool. But it's also because consumer goods have a good century head start in the packaging game. Packaging features that have long been recognized as important parts of how consumers buy and use products have only recently gained serious attention in the software world.

Inform

We begin by turning the discussion to how packaging informs. This is inevitably wrapped up with the broader ways in which packaging communicates and even becomes part of how people think about, feel about, and use a product. But we'll start with those aspects of communication that are most about communicating facts rather than building more subjective experiences.

Informational packaging was originally pretty bare-bones. A bag might have "flour" printed on it or a soap wrapper the manufacturer's name.



General store in US c. 1900. Note the relatively limited amount of labeling.

The object being sold might have been expected to do its own communicating. This largely remains the case at a farmers market, produce section, or butcher today. A price is likely on display and there may be sign telling you the variety of tomato or cut of meat on offer. But not much else.

Historically, selling was also largely an interactive exchange between a buyer and a seller. A bazaar is the classic example, but even a nineteenth century general store usually involved a customer asking for and receiving goods through an intermediary, the shopkeeper. To the extent that buyers needed additional information, they asked.

This model began to change in the early twentieth century.

The shift to self-service

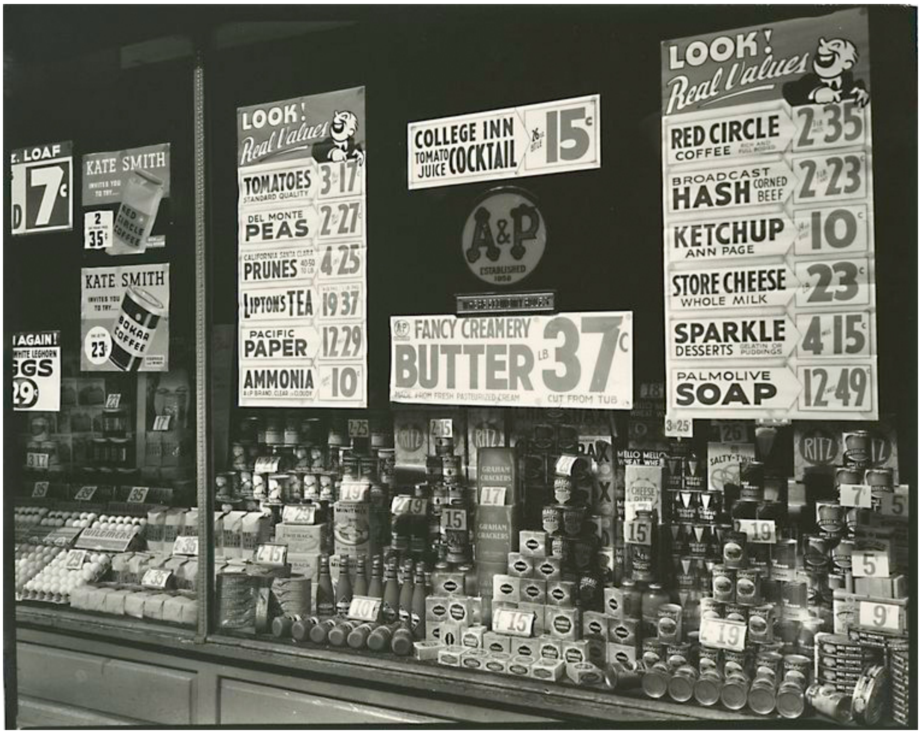
Piggly Wiggly, founded in 1916 in Memphis, Tennessee by Clarence Saunders, is often credited with being the first true self-service grocery store. At the time of its founding, grocery stores did not allow their customers to gather their own goods. Instead, a customer would give a list of items to a clerk, who would then go through the store himself, gathering them. Piggly Wiggly introduced the innovation of allowing customers to gather their own goods. This cut costs, allowing for lower prices.[37]

Chain store retail was taking off at about the same time with the Great Atlantic and Pacific Tea Company (later A&P), established in 1859, and other small, regional players including Piggly Wiggly. In the late 1930s, A&P began consolidating its thousands of small stores into larger supermarkets, often replacing as many as five or six stores with one large, new one. Similar transformations occurred among all the major players; in fact, most national chains of the time

[37] http://www.groceteria.com/about/a-quick-history-of-the-supermarket/

saw their store counts peak around 1935 and then decline sharply through consolidation. This consolidation coincided with the introduction of self-service at A&P in 1936.

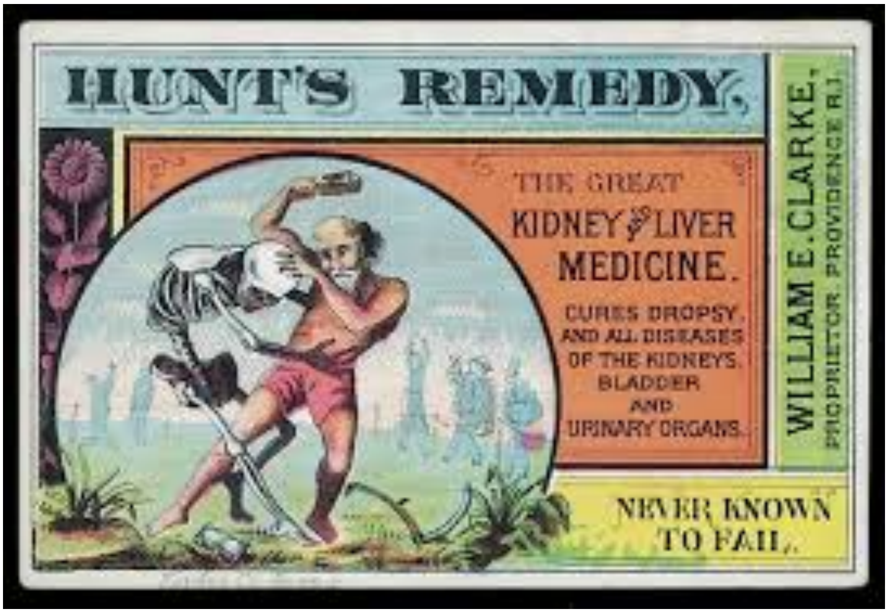
Compare photographs of food stores or general stores before and after self-service and the difference is striking. In the after photos, the boxes and cans are designed to grab the consumer's attention both graphically and with information about their content.


LOOK! Real Values
TOMATOES 3-17
DEL MONTE PEAS 2-27
CALIFORNIA SANTA CLARA PRUNES 4-25
LIPTONS TEA 19 37
PACIFIC PAPER 12-29
AMMONIA 10
COLLEGE INN TOMATO JUICE COCKTAIL 15¢
A&P ESTABLISHED 1859
FANCY CREAMERY BUTTER LB 37¢
MADE FROM FRESH PASTEURIZED CREAM CUT FROM TUB
LOOK! Real Values
RED CIRCLE COFFEE 2-35
BROADCAST HASH CORNED BEEF 2-23
KETCHUP ANN PAGE 10¢
STORE CHEESE WHOLE MILK 23¢
SPARKLE DESSERTS GELATIN OR PUDDINGS 4-15¢
PALMOLIVE SOAP 12-49
KATE SMITH INVITES YOU TO TRY... RED CIRCLE COFFEE 2 35¢
KATE SMITH INVITES YOU TO TRY... BOKAR COFFEE 23¢
LOAF 7¢
AGAIN! WHITE LEGHORN EGGS 29¢
GRAHAM CRACKERS
RITZ

A&P, 246 Third Avenue, Manhattan, 1936. Note the prominent ads for A&P's private brands. Source: Wikimedia, released into the public domain by the New York Public Library.

Of course, it helps if the information being communicated is true.

For example, a patent medicine like Hunt's Remedy presented itself as the "Great Kidney and Liver Medicine" that "cures dropsy and all diseases of the kidney, bladder, and urinary organs." It was "never known to fail." Norman's Snake Oil liniment promised "instantaneous relief" and to cure "all aches and pains with the strength of a thousand snakes."


HUNT'S REMEDY.
THE GREAT
KIDNEY & LIVER
MEDICINE.
CURES DROPSY, AND ALL DISEASES OF THE KIDNEYS, BLADDER AND URINARY ORGANS.
WILLIAM E. CLARKE,
PROPRIETOR, PROVIDENCE R.I.
NEVER KNOWN TO FAIL.

The Pure Food and Drug Act of 1906 was the first of a series of consumer protection laws enacted by the US Congress in the twentieth century; it led to the creation of the Food and Drug Administration. Among other purposes the law was intended to ban mislabeled food and drug products. It also required that active ingredients be placed on the label of a drug's packaging and that drugs could not fall below established purity levels.

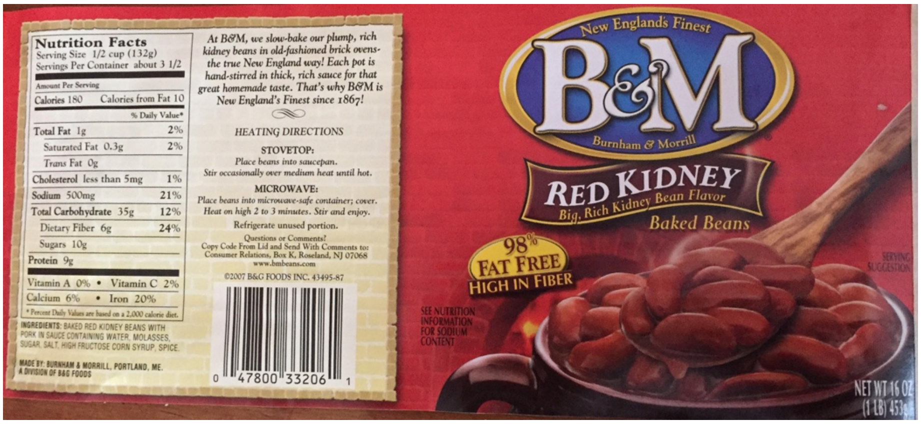
However, in United States v. Johnson in 1911, the United States Supreme Court ruled that the misbranding provisions of the Pure Food and Drug Act of 1906 did not pertain to false curative or therapeutic statements; rather, it only prohibited false statements as to the identity of the drug. Congress responded in 1912 with the Sherley Amendments, which prohibited false and fraudulent claims of health benefits.

In their own way, computer software products have made almost equally outrageous promises. Given that, in theory, missing functionality is just an update away, it can be tempting to make claims that reflect aspirations more than they do reality. Furthermore, especially before the widespread use of open source made it easier to test and examine software, it could also be expensive and time-consuming to figure out if products worked as advertised.

A familiar example of government-mandated information on packaging today is the nutrition facts label. In the US, this was mandated by the Food and Drug Administration in 1990. In addition to the nutrition label, products may display certain nutrition information or health claims on packaging. These health claims are only allowed by the FDA for eight diet and health relationships based on proven scientific evidence.

Packaging can also convey information about what a product is for, how to use it (and how not to use it!), claims relative to other products, and which other products from the company you might like to use with this one.

Take, for example, a box of Barilla spaghetti sitting on a shelf. One panel tells us how to “get the best from your pasta, cooking the Italian way” in three steps. Another panel tells us what’s inside and the net weight of the contents. The flip side advertises claims such as “part of a healthy diet” and “non-GMO ingredients.” A stamp informs that the contents of this box are best used by January 2019 and includes some identifying information that is probably relevant to the company for recalls and other purposes.



Nutrition Facts
Serving Size 1/2 cup (132g)
Servings Per Container about 3 1/2

Amount Per Serving

Calories 180	Calories from Fat 10
	% Daily Value*
Total Fat 1g	2%
Saturated Fat 0.3g	2%
Trans Fat 0g	
Cholesterol less than 5mg	1%
Sodium 500mg	21%
Total Carbohydrate 35g	12%
Dietary Fiber 6g	24%
Sugars 10g	
Protein 9g	
Vitamin A 0% • Vitamin C 2%	
Calcium 6% • Iron 20%	

* Percent Daily Values are based on a 2,000 calorie diet.

INGREDIENTS: BAKED RED KIDNEY BEANS WITH PORK IN SAUCE CONTAINING WATER, MOLASSES, SUGAR, SALT, HIGH FRUCTOSE CORN SYRUP, SPICE.

MADE BY: BURNHAM & MORRILL, PORTLAND, ME. A DIVISION OF B&G FOODS

At B&M, we slow-bake our plump, rich kidney beans in old-fashioned brick ovens- the true New England way! Each pot is hand-stirred in thick, rich sauce for that great homemade taste. That's why B&M is New England's Finest since 1867!

HEATING DIRECTIONS

STOVETOP:
Place beans into saucepan. Stir occasionally over medium heat until hot.

MICROWAVE:
Place beans into microwave-safe container; cover. Heat on high 2 to 3 minutes. Stir and enjoy.

Refrigerate unused portion.

Questions or Comments?
Copy Code From Lid and Send With Comments to: Consumer Relations, Box K, Roseland, NJ 07068
www.bmbeans.com

©2007 B&G FOODS INC. 43495-87

0 47800 33206 1

Typical canned food label showing branding, informational content, instructions, ingredients, nutrition facts label, and UPC.

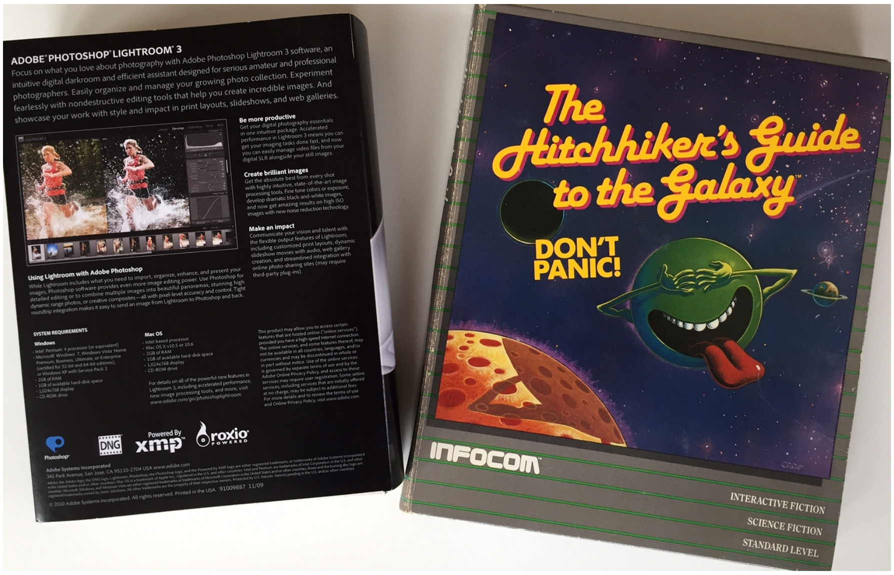
In years past, we'd also expect to have seen some part of a human-readable price label added by a retailer. Today, though, that information is often on the shelf rather than the individual box or can.

In part, that's because there's now a barcode. This is still information, of course. But it's information that is used as part of the retail system rather than by the consumer directly.

As Margalit Fox wrote in The New York Times in 2011: “On a summer morning in 1974, a man in Ohio bought a package of chewing gum and the whole world changed. At 8:01 a.m. on June 26 of that year, a 10-pack of Wrigley's Juicy Fruit gum slid down a conveyor belt and past an optical scanner. The scanner beeped, and the cash register understood, faithfully ringing up 67 cents. That purchase, at a Marsh Supermarket in Troy, Ohio, was the first anywhere to be rung up using a bar code." (To be a bit more precise, this was the first commercial use of the Universal Product Code (UPC) specifically.)

Software packaging can be directly informational as well whether the information is for a human looking at a package or packaging system or (as is increasingly the case), it's in a form that can be interpreted and acted upon by the software itself.

The trivial example of human-readable information in software packaging comes from shrink-wrapped software boxes. A typical box would tell you what sort of computer the software was written for and minimum specs for the hardware and operating system. The generally expensive software of the early microcomputer era would also throw in manuals, reference cards, and other content that would help people use the program stored on the enclosed diskettes. Early PC software boxes were often designed to stand out from the competition but, over time, retailer demands led to more standardized sizes and shapes.



Better information through bits

However, the more interesting discussion concerns how the bits themselves can be packaged to convey information describing the software, what's needed to run it, and how to install it. The trend over time has been to make software more self-contained and enable the informational content to take direct action rather than simply being a set of instructions for a human to follow.

An early step down this path is the archive utilities that have existed in many operating systems. In the Unix world, the best known is tar, an archive format that collects files, directories, and other file system objects into a single stream of bytes, which can then be written out as a file. The tar utility(as mentioned earlier in the context of container layers) was first introduced in the Seventh Edition of Unix in January 1979, replacing the tp program. Like most other archive utilities, tar could also compress the contents of the archive, thereby reducing the amount of disk space required to store it and the time needed to transmit it over a phone line or network.

In the PC world, the first widely-known and used archive utility to also compress files was ARC, written by Thom Henderson of System Enhancement Associates (SEA) in 1985. ARC was especially popular on hobbyist bulletin board systems (BBS), both because it packaged all the files associated with a program into one download and because compression reduced the time needed to download files using modems on telephone lines that could only transmit a few hundred characters per second. A few years later, after a nasty and controversial lawsuit, ARC largely gave way to PKWare's ZIP format, developed by Philip Katz using some of SEA's code, which had been made public but not under an open source license. The ZIP format remains in wide use today although programs are more likely to be packaged up in different ways, as we shall see.

Archive utilities were fine as far as they went. They put all the necessary files in one place and then transferred them to disk in a structured way so that they were laid down in a way the main program expected when it was run. For example, they might place documentation in a specific directory rather than putting all of a program's files in a big jumble. However, unpacking an archive did nothing to customize the installation for a particular system or a particular user. That required an installer.

Installers and package managers

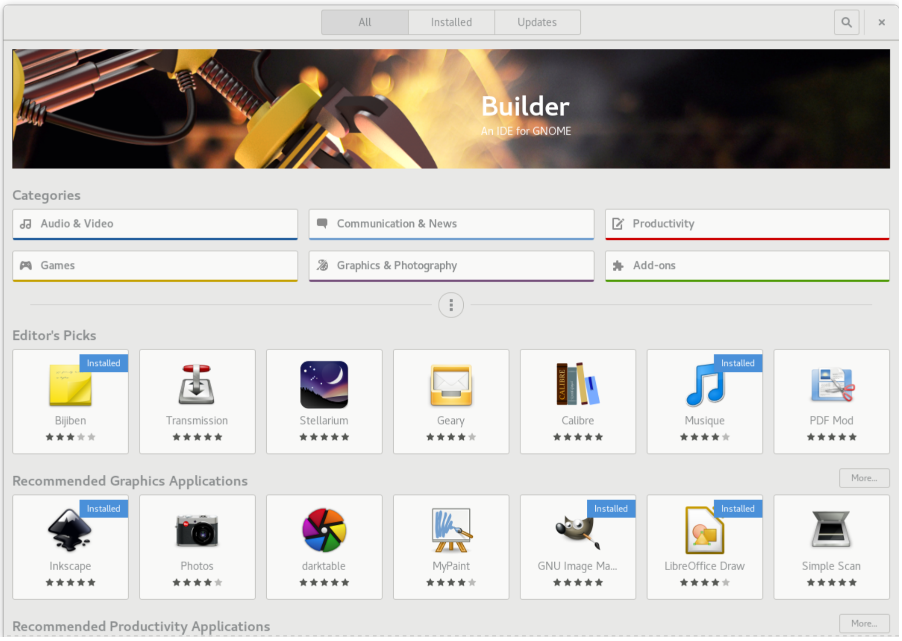
Installers have often been one-off affairs. There's been some standardization within various operating systems, Microsoft Windows in particular. But installers have often failed to provide a consistent experience when loading a program onto a system, a consistent way to determine and load the software on which a program depended, or a consistent way to update a program over time. Traditional installers were (and are) often something of a hack.

Package managers were the response, mostly on systems running Linux, to automate the process of installing, upgrading, configuring, and removing programs in a consistent manner. Originally written by Red Hat's Erik Troan and Marc Ewing in 1997, RPM is an early example. Other examples include yum, and its successor DNF, for RPM-based distributions,[38] and apt, for Debian-based distributions like Ubuntu.

Yum, and its successor DNF can resolve dependencies and perform other checks on package installations (as can apt). Yum uses the RPM file format. When yum (or DNF) finds dependencies that are not installed it can source those dependencies from an online

[38] RPM is itself technically a package manager but yum and DNF build on it to provide more sophisticated package management features such as resolving dependencies and taking the appropriate actions in response.

repository ("repo") and install them before installing the desired package. Yum/DNF also has a graphical user interface that provides an "App Store" type experience with built-in search capabilities and update notifications.



The "Software" GUI that provides App Store style interface over DNF on Fedora.

To their detriment, neither Microsoft Windows nor Mac OS X ever introduced a package manager although others have written package managers for OS X. (Most notably, Homebrew and MacPorts. Because OS X is built on a BSD Unix foundation, it's amenable to package management in the Linux vein.) However, today, mobile app stores such as Apple's can also be thought of as a form of package management although they're based on a very different model that is conceptually more related to the containers model.

Package managers were and continue to be an extremely useful tool for managing Linux systems. But with the availability of both container technology and new approaches to configuration management, it's now possible to embed information that makes installing and running software an even more consistent and more automated experience.

Dockerfiles

The revolution that the docker project originally brought to the container technology space was largely in two areas. First, as mentioned earlier, docker created the de facto standards for the runtime and layered image format that were later rolled into OCI.

The docker project also contributed the notion of a Dockerfile and a comprehensive CLI to interact with it. A Dockerfile describes how you would build a new image using a series of commands. Dockerfiles are not necessary for creating an OCI-compliant image file but they're a common mechanism for doing so.

In this way, information needed to run an application or service can be embedded into a container together with the minimum software layers that they need to perform a task. Containers are intended to house extremely lean application stacks.

A Dockerfile (or other OCI-compliant technology) provides a means for automating the building and runtime certification of containers and their images. This in turn enhances automation. This is particularly important because automation is central to modern, agile approaches to software development and operations including DevOps practices.

Alternatives to docker and Dockerfiles

There are two potential issues with using docker and a Dockerfile.

1. Many Dockerfiles depend on package management tools to install new packages into the container. This requires the base image to have the package manager(s) as part of the base image. This means that the base image is larger than it needs to be and the resulting container has packages that it does not require to run.
2. The `docker build` command requires the docker runtime to start the base container and build the image layers inside the container and then persist it into a tar file.

As a result of this overhead, other efforts adhering to the OCI specification have been developed. Project Atomic's Buildah is such a project. It uses the underlying container storage to build the image and does not require a runtime. As a result, it also uses the host's package manager(s) to build the image and therefore the resulting images can be much smaller while still meeting the OCI spec.

The larger point here is that OCI standardization has freed up a lot of innovation. Much of the image building, registry pull and push services, and container runtime service are now automated by higher level tools like OpenShift. Though it's still useful to use or experiment with command line container tooling from an educational and trouble-shooting perspective, automation ends up hiding a lot of this detail when you are working with containers at scale in an enterprise environment.

Configuration management and playbooks

Once you move into highly automated and highly scalable CI/CD environments, efficiency and velocity become especially important. So long as container image and runtime requirements are met, organizations have a lot of flexibility to pick technologies that meet efficiency and velocity and security requirements—without losing the benefits of standardized containers.

Automation and the delivery of complete applications to computer systems didn’t start with containers nor does it end there today. As our colleague Mark Lamourine said in a podcast:[40]

> It started out when I was the young cub sysadmin, we'd have a set of manual procedures that started out as things in our head: Set the network, set resolv.conf, set the hostname, make sure time services were running.
>
> When you only had a short list of these things, it wasn't really a big deal. You'd go to each machine, you'd spend 15 minutes making it fit into your network, and then you'd hand it off to some developer or user.
>
> Over time, we realized that we were doing an awful lot of this and we were hiring lots of people to do this, so we needed to write scripts to do it. Eventually, people started writing configuration management systems, starting with Mark Burgess and CFEngine.
>
> The idea was that we were doing these tasks manually. We started automating them, but we were automating them in a custom way.
>
> Then people recognized patterns and said, “We can do this. There's a pattern here that we can automate, that we can take one step higher.” That led to these various systems which would make your machines work a certain way.

[40] http://bitmason.blogspot.com/2015/02/podcast-configuration-manangement-with.html

Over time the configuration management space evolved; different systems followed different philosophies and were tailored to the approaches of different types of users. For example, Puppet is generally considered to be attuned to the historical practices of system administrators while Chef is more aligned with a developer mindset.

However, the same changing software patterns that have helped popularize containers are also changing complementary tools like configuration management. This stems, in part, from the growing scale of many software deployments. The shift from monolithic applications that are long-lived and monolithic to short-lived microservices, as we discussed earlier, is another important factor.

One example of a more modern approach to automation is provided by Ansible, which has become extremely popular. It's popular for a variety of reasons, not least of which is that it's easy to get productive quickly. It's also "agentless"—which is to say that Ansible does not require installing any components on a managed host. Instead, Ansible relies on the existing secure shell technology (ssh). In this way, Ansible can reach out to a host through the secure shell and run any command.

Ansible also allows vendors to create plugins that take advantage of any API (application programming interface) or CLI that their technology uses. In this way, Ansible is extensible while still maintaining a zero footprint on the managed host, a good match for lightweight application components. Vendor and technology plugins provide a way to use the Ansible playbook language to take advantage of APIs to generate scripts efficiently.

Ansible's playbooks provide a language to describe the policy for successful configuration and deployment of remote systems. In this way, a playbook can be used to configure and deploy thousands of remote hosts at the click of a button. For example, in 2015 in San Francisco, Riot Games described how they used Ansible to deploy an entire gaming data center, including databases and virtual networks, in four minutes. This was down from five weeks![41]

Ansible also provides a mechanism to evaluate the success or failure of a particular task in the playbook. A failed task can result in the playbook halting and providing feedback to the user about which host failed. In this way a playbook can enforce a policy (such as who has access to an application) across multiple hosts.

As a result of container standardization, Ansible has also been able to focus on using automation to help build and deploy container technology. Ansible Container will build OCI-compliant containers without depending on a Dockerfile.

With respect to modern packaging, this shows how, thanks to standardization and by building on existing automation approaches, further innovations around automation have increased efficiency and velocity even further.

Minimal Optimized Container Hosts

Another byproduct of containerization has been the evolution of container-friendly operating systems.

A container includes the entire stack of binaries and libraries that is required by an application and only requires the host's kernel and a very minimal set of services to run. Therefore, why use a full Linux operating system distribution? A developer may need to have a full distribution installed on their laptop. But once a container moves into a CI/CD process, there is no need for a distribution on the host

[41] http://www.esg-global.com/blog/network-automation-joy

containing more than the bare minimum set of services. Anything else just adds overhead (and more ongoing maintenance burden).

To be clear, you still need the full capabilities of the Linux kernel. But you don't want all of the other software that ships in a typical distribution.

Consider, again, container ships. Once you have standardized on a shipping container and you then want to transport those containers over vast oceans, you develop an optimized ship that can transport as many of those containers as possible. You require engines, fuel, a pilot cockpit, and some comfortable living quarters and emergency lifeboats for the sailors. But beyond that there is little point in burdening the ship with any other amenities. The purpose is to efficiently move as many containers as possible. (The largest container ship sailing today, the MOL Triumph, has a capacity of a whopping 20,170 TEU.[42])

A distribution like Red Hat Enterprise Linux Atomic Host is an example of a minimal-footprint operating system optimized for containers. It's built from Red Hat Enterprise Linux but all applications and tools run inside containers and it's tuned and optimized for running containers. Atomic is also immutable, in that a deployed Atomic image cannot be added to or changed. Rather, any update is an all or nothing update and the host is restarted when the update is complete. This reflects another aspect of new application architectures in which individual components are considered disposable and don't get tweaked and repaired throughout an extended life cycle.

[42] Twenty-foot equivalent unit. This is the standard unit used to specify container ship capacity although, in practice, double-length 40 foot containers have found wider acceptance, as they're the size of a typical semi-trailer truck. This length is within the limits of national road regulations in many countries, requiring no special permission.

Container registries

As with package managers, container platforms and container users require a service to provide containers for download to a host. These services are called a container registry. We discussed registries earlier in the context of securing the software supply chain. The registry is the service from which a host downloads a container image and the repository is the place on the host where container images are stored and started from.

The `docker pull` command is used to download an image from a registry to a host repository and the `docker push` command is used to upload a local container image from a repository to a registry. (This terminology is closely related to that used by modern source control systems such as git.) The project Skopeo was originally developed to just query image metadata (the json file) from a registry, but has since developed into a full registry pull/push utility similar to docker's pull/push.

In this way developers can upload new images, with new tags, to a registry for other developers or testers to pull down. Docker Inc. provides a default registry, also known as the Docker hub, where users around the world can pull and push images. However, if there are concerns over downloading from the public internet, then a private local registry can be used instead. A private registry provides more assurance over the image quality and certification than a registry which anyone can upload to can provide. Download performance can also be better over a private network.

There's limited standardization of registries overall, but distribution is another area of standardization that the OCI is considering tackling.

As container platforms have evolved to provide faster and more efficient automation, there have also been other advances in container registries. For example, built-in certification and image scanning of some registries give the users of that registry confidence in the contents that they’re planning to use.

Creating an Experience

Everything we've discussed up to this point has been about been about enabling and simplifying but we've been effectively dancing around the edges of the central goal, the central desire of the consumer, which is a better experience. Of course, packaging can make the experience, the transaction, better for the vendor too.

But what does better mean?

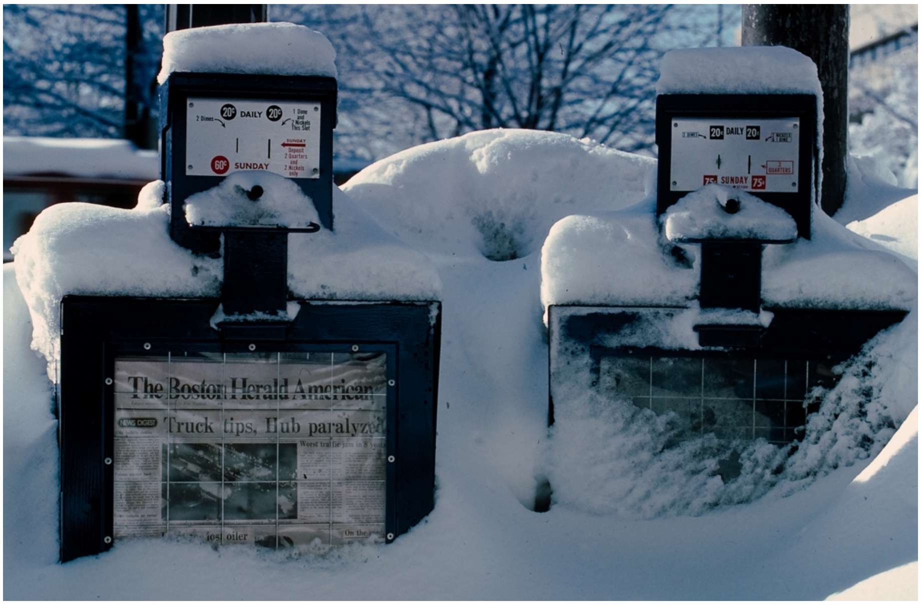
Bundling

Some aspects of "better" are certainly from the perspective of the seller. The idea of bundling, in some respects a superset of product packaging, is one example of this. By making the customer an all or nothing offer, a bundle prevents a customer from picking and choosing parts they don't want and negotiating individual line items.

We touched on this earlier in the context of products. Auto makers have become masters of the bundling game when it comes to options. You want those heated leather seats? Sure. But you need to take the alloy rims and the upgraded trim kit too.

But bundling is really a broader concept and there's perhaps no more canonical example historically than newspapers.

Newspapers bundle various news topics like syndicated and local news, sports, and political reporting, along with advertising, classifieds, weather, comic strips, shopping coupons, and more. Many of the economic woes of the newspaper can be traced to the splitting of this bundle. Craigslist took over the classifieds—and made them mostly free. Online severed the connection between news and local ads. While ads run online as well, the economics are something along the lines of print dollars devalued to digital dimes.



Newspapers are a classic example of a bundle that creates a product from parts that may not be individually viable. Source: Gordon Haff.

As NYU professor Clay Shirky wrote in 2008:[46]

> For a long time, longer than anyone in the newspaper business has been alive in fact, print journalism has been intertwined with these economics. The expense of printing created an environment where Wal-Mart was willing to subsidize the Baghdad bureau. This wasn't because of any deep link between advertising and reporting, nor was it about any real desire on the part of Wal-Mart to have their marketing budget go to international correspondents. It was just an accident. Advertisers had little choice other than to have their money used that way, since they didn't really have any other vehicle for display ads.

[46] https://www.edge.org/conversation/clay_shirky-newspapers-and-thinking-the-unthinkable

Over the years, many tech companies have attempted to force customers to buy a bundle. Wanted phone service from the old AT&T (and it's not like you had a choice)? You had to rent a phone from the local Bell operating company. You could have it in any color you wanted so long as that color was black, but it *was* solidly built.

In computer and related office equipment businesses, the bundle was typically some combination of hardware, software, services, and supplies. One of the primary motivations was to prevent a competitor from cherry-picking some aspect of your business to compete against. However, bundles also made possible less obvious subsidies and pricing models. For example, when IBM and Xerox tied the sale of supplies like punched cards and paper to their leased machines, this effectively gave them a way to meter usage and price discriminate between high volume users and low volume ones.

Tying has played a part in a number of the tech industry's antitrust cases. That's because, as in the case of the local newspaper, a dominant position in some market greatly increases the power of a company to enforce a bundle without worrying about what competition might do in response.

The central issue of United States v. Microsoft in 2001 was whether Microsoft was allowed to bundle its flagship Internet Explorer browser software with its Microsoft Windows operating system. Bundling them together was alleged to have been responsible for Microsoft's then-victory in the browser wars as every Windows system came with a copy of Internet Explorer out of the box. (The outcome of the case was complicated but Microsoft eventually agreed to a settlement.)

Other examples include Data General vs. Digidyne in which Data General, then a maker of minicomputers (what we'd call servers today), was forced to sell its RDOS operating system to Digidyne to run on its "clone" hardware.

Most recently, there has been the ongoing squabble over digital rights management (DRM) in printer cartridges. This is an attempt by printer manufacturers to limit the use of third-party ink cartridges in their printers. This is one of the clearest examples of cross-subsidies. Low-end printers are sold at or below cost. They're profitable only because of ink sales—which, of course, the manufacturer doesn't get if you buy someone else's ink.

But there's another view of bundling that ties back to product packaging and user experience.

Bundles, like other aspects of packaging, are prescriptive. They can be seen as a response to *The Paradox of Choice*, a 2004 book by American psychologist Barry Schwartz, in which he argues that consumers don't seem to be benefitting psychologically from all their autonomy and freedom of choice. Whether or not one accepts Schwartz' disputed hypothesis, it's certainly the case that technology options can sometimes seem to proliferate endlessly with less and less real benefit to choosing one tech over another.

Indeed, from the perspective of a newspaper or magazine reader, one of the advantages of certain aspects of the newspaper bundle is that it delivers a curated news experience for one predictable price. A limited number of publications—including *The Wall Street Journal*, *The New York Times*, and *The Economist*—have demonstrated that there's still some market for this even in an online world.

Indeed, some news organizations such as *The New York Times*, which has achieved some success with digital subscriptions, are experimenting with new forms of bundling. In its 25.03 issue in early 2017, *Wired* magazine described how the *Times* has been developing new products such as Cooking, Real Estate, and Watching as part of its Beta Group. (The acquisition of the gadget review site Wirecutter made for the newest product to be brought into Beta.) Collectively, it's a form of bundling for a digital subscription age.

There are numerous other examples of bundles whose components are not as attractive to some consumers in their fully disaggregated state.

Some bundles of financial instruments have gotten a bad rep for good reason. In part it was poorly structured bundles of loans known as collateralized debt obligations (CDOs) that exacerbated the 2008 sub-prime mortgage crisis. The complexity of these bundles was one factor that obscured how risky they, in fact, were.

However, bundles are ubiquitous throughout the financial industry because they can also reduce risk or otherwise hedge against unforeseen events. Mutual funds are bundles of individual stocks, bonds, and other investments. They allow investors (for a fee) to buy into a more diversified portfolio than they would otherwise be able to. Other instruments allow airlines to hedge against fuel price increases. (Airlines generally prefer to focus on being profitable as an airline, not by speculating on oil prices.) Interest rate swaps can better line up incoming and outgoing cash flows. For example, a company that has fixed rate loans and floating rate investments might desire a financial instrument that locks in their investment income at a fixed rate.

Bundling can also be another aspect of delivering an integrated and tested experience. The manufacturer of those DRMd printer cartridges is being more than a bit disingenuous when they say that they're doing it for your own good. Nonetheless, having visibility and control over the supply chain and manufacturing of all the components that will be used together as part of a product and process reduces the likelihood that sub-par parts will make their way in. (Though you may pay a premium for this assurance.)

Furthermore, bundling simplifies the transaction and the support after the transaction. To return to Clay Shirky and newspapers, a la carte pricing models for unbundled short-form writing, such as a single article or a blog post, have proven elusive. Micropayments in the "give me a nickel to read this story" vein have failed time and time again. Way back in 2000, Shirky argued that this was because "users want predictable and simple pricing. Micropayments, meanwhile, waste the users' mental effort in order to conserve cheap resources, by creating many tiny, unpredictable transactions. Micropayments thus create in the mind of the user both anxiety and confusion, characteristics that users have not heretofore been known to actively seek out."[47] This transaction cost argument sounds a lot like the paradox of choice.

But there are no formulas for bundles and pricing. People say that they hate being "nickeled and dimed." Yet, they may not like that monthly subscription bill for a service they don't use much either. Consumers widely grouse about cable bills that include hundreds of channels that they never watch. Start adding up streaming services that need to be individually paid for and lack a common interface and that doesn't seem ideal either.

The unboxing experience

Ultimately, whether it's software or something else, there's a need that's being fulfilled and the packaging should be in service of that goal. But that's not to say that packaging is purely about getting a

[47] http://www.openp2p.com/pub/a/p2p/2000/12/19/micropayments.html

consumer to some goal as efficiently as possible—though that's certainly part of it.

Signs of the evolution of packaging from the utilitarian to the experiential are everywhere.

Unbox a computer a couple of decades ago and, if you were lucky, you might find a sheet of paper easily identifiable as a "Quick Start" guide. (Which itself was an improvement over simply needing a field engineer to swing by.)

Today the unboxing experience of consumer goods like Apple's iPhone has become almost a cliché, but it's no less real for that. In the words of Grant Wenzlau, the creative strategist at Day One Agency, "Packaging is no longer simply about packaging the object—it is about the unboxing experience and art directing. This is where the process starts for designers today: you work backward from the Instagram image to the unboxing moment to the design that serves it."

The idea of creating an experience around acquiring a product isn't new.

One of the clear antecedents in retail comes from Harry Gordon Selfridge, the American retail magnate who founded the London-based department store Selfridges.

Selfridge promoted the notion of shopping for pleasure rather than necessity (at his Oxford Street store of course.) As Erika Rappaport writes: "Gordon Selfridge marketed his new store by promoting shopping as a delightful and respectable middle-class female pastime… In writing about the store's opening, [one] paper's reporter loudly proclaimed that, at Selfridge's, 'Shopping' had become an 'Amusement.' Whether imagined as an absolute need, a luxurious treat, a housewife's duty, or a feminist demand, shopping was always a pleasure."[48]

A Selfridges Christmas display. Shopping as experience. Source: Selfridges.

Selfridge's housed elegant restaurants with modest prices, a library, reading and writing rooms, special reception rooms for French, German, American and "Colonial" customers, a First Aid Room, and a Silence Room, with soft lights, deep chairs, and double-glazing, all intended to keep customers in the store for as long as possible. Staff members were taught to be on hand to assist customers, but not too aggressively, and to sell the merchandise.[49]

Over time, the idea of thinking about user experience more broadly took hold. One could point to Frederick Winslow Taylor's early twentieth century research into how workers interact with their tools as a precursor to the science behind how we think about user experience today. Peter Drucker, who once graced the cover of

[48] The Gender and Consumer Culture Reader.
[49] https://en.wikipedia.org/wiki/Harry_Gordon_Selfridge

Business Week as “the man who invented management,” wrote that Taylor “was the first man in recorded history who deemed work deserving of systematic observation and study. On Taylor's ‘scientific management’ rests, above all, the tremendous surge of affluence in the last seventy-five years which has lifted the working masses in the developed countries well above any level recorded before, even for the well-to-do.”

Beyond interfaces to experiences

The modern focus on user experience is often connected to Donald Norman whose 1986 *The Design of Everyday Things* is a classic of the field. However, Norman himself says that earlier user experience thinking was too narrow in scope. Writing in the expanded 2013 edition of his earlier book, he writes: “The first edition of the book focused upon making products understandable and usable. The total experience of a product covers much more than its usability: aesthetics, pleasure, and fun play critically important roles. There was no discussion of pleasure, enjoyment, or emotion. Emotion is so important that I wrote an entire book, *Emotional Design*, about the role it plays in design.”

Software has a (deserved) reputation for historically paying scant heed to usability. But especially once graphical user interfaces became widespread, designers started paying more attention to user interface (UI) design and then user experience (UX) more broadly. One can even observe the evolution from UI to UX through the lens of book titles. In 1992, Bruce Tognazzini, then Human Interface Evangelist at Apple, published *Tog on Interface* which mostly focused on things like learning curves and consistency. Fifteen years later, Bill Buxton of Microsoft published *Sketching User Experiences,* which focuses on higher-level attributes:

> Despite the technocratic and materialistic bias of our culture, it is ultimately experiences that we are designing, not things. Yes, physical objects are often the most tangible and visible outcomes of design, but their primary function is to engage us in an experience—an experience that is largely shaped by the affordances and character embedded into the product itself. Obviously, aesthetics and functionality play an important role in all of this.

Part of this experience is rooted in how easily software is acquired, prepared for use, and operated for however long it's needed. As analyst Stephen O'Grady wrote in his 2012 post "Do Not Underestimate the Power of Convenience:"[50]

> One of the biggest challenges for vendors built around traditional procurement patterns is their tendency to undervalue convenience. Developers, in general, respond to very different incentives than do their executive purchasing counterparts. Where organizational buyers tend to be less price sensitive and more focused on issues relating to reliability and manageability, as one example, individual developers tend to be more concerned with cost and availability—convenience, in other words.

One of the most recent software trends, nascent as of this writing, is what goes by the (unfortunate) moniker "serverless computing."[51] This abstracts away underlying infrastructure to an even greater degree than containers, allowing for suitable functions—think encoding an uploaded video file—to run in response to events and other triggers. Most associated with Lambda at Amazon Web Services currently, a variety of open source projects in this space

[50] http://redmonk.com/sogrady/2012/12/19/convenience/
[51] Of course, there's still a server. Given the choice, we much prefer Functions-as-a-Service (FaaS), consistent with cloud computing service delivery terminology more broadly.

such as OpenWhisk are also underway. The overall goal can be thought of as almost making the packaging invisible while letting developers implement an idea with as little friction as possible. It's a logical extension to container platform concepts that allows users to choose more prescriptive but more convenient bundles of functionality.

The central tension

This brings us to one of the central tensions in the IT industry today. On the one hand, there's the innovation taking place in open source in all its sometimes unruly and rough-around-the-edges glory. On the other hand, there are the generally more packaged, curated, and polished offerings from what we'll call generically "the cloud" whether as complete Software-as-a-Service applications or as more discrete cloud services such as storage.

This isn't a new tension. Just a little over a year after Amazon, still the leader in public cloud services today, announced their first iteration of Amazon Web Services (AWS) in 2006, Gordon wrote a research note titled "The Cloud vs. Open Source."

> Some of the concepts within AWS had existed previously. S3 resembled the storage service providers of the dot-com era. EC2 bore more than a passing resemblance to Sun Microsystem's much-hyped Sun Grid Compute Utility—although that was based on physical servers rather than AWS' virtual infrastructure. But Amazon succeeded where those others had not through a combination of scale, low pricing, embracing new lightweight Web protocols, and an aggressive focus on continually rolling out new services and new capabilities.

It probably didn't hurt either that AWS rolled out around the dawn of the second great Internet boom. This one distinguished itself from the first one in part by far less investor appetite for huge outlays of up-front capital spending on rooms full of computers, disks, and networking gear. In this startup climate, the availability of cheap pay-per-use compute capacity was extremely attractive.

Some of that writing seems a bit off today. It didn't really foresee the degree to which cloud service models would inspire whole new categories of computer software. (Including containers which, at that time, were mostly just a little-used alternative to hardware virtualization.)

The tension between the freedom and flexibility of open source and the convenience of the cloud. Source: Illuminata.

But it did get a few things right that remain relevant.

First (and probably controversially at the time) was that it downplayed the importance of open source licensing that requires modifications to be contributed back to the commons under some circumstances. Gordon wrote:

> Such a worldview implicitly assumes that copyleft[52] is the only reason that Open Source users contribute back their

[52] A copyleft license requires that if changes are made to a program's code, and the changed program is distributed outside an organization, the source code containing the changes must likewise be distributed. Permissive licenses don't.

> enhancements. Copyleft may or may not have played a major role in the rise of Open Source. Certainly, the GPL has long been the most common Open Source license, used by Linux, GNU, and many others. However, the BSD license—which does not require that code changes be made available—is also widely used. It's an interesting historical debate whether the ultimate impact of Linux was far greater than the BSD operating system because of license differences, or because of other reasons—of which there were many. In any case, Open Source does not begin and end with the GPL and copyleft.

And, indeed we've seen a general trend toward permissive licenses such as the Apache Software License[53] and very limited take-up of licenses that close some cloud software delivery "loopholes" such as the Affero GPL. This shift reflects less concern about preventing free-riders and more concern about growing communities.

The Eclipse Foundation's Ian Skerrett puts it this way: "I claim all these projects use a permissive license to get as many users and adopters, to encourage potential contributions. They aren't worried about trying to force anyone. You can't force anyone to contribute to your project; you can only limit your community through a restrictive license."

Which brings us to Gordon's next point which remains germane today:

> Indeed, focusing too narrowly on Open Source in a Cloud Computing world is counterproductive. Source code may matter, or it may not, depending upon the circumstances. But it's the many other aspects of Open Source development

[53] Matthew Aslett of market researcher 451 Group wrote in 2011 that: "2010 was the first year in which there were more companies formed around projects with non-copyleft licenses than with strong copyleft licenses."

> (community, extensibility) or Open Source principles (portability of data, open formats) that matter far more.

Open source code allows organizations to collaborate with each other. It's not sufficient. It's an enabler but collaboration happens because openness exists across many dimensions within an environment where people can work together.

Without a viable, independent community, it's hard to realize the collaborative potential of open source. Delivering the most innovation means having the right structures and organization in place to fully take advantage of the open source development model.

There's no single approach to fostering communities. The best approach in any given case to engaging with and governing a community will depend on the nature of the project. Who is contributing? What are the project's goals? What business or licensing constraints are there? These and many other factors will affect governance structure, as well as copyright, trademark, and licensing decisions.

Open standards, or protocols and formats that are moving toward standardization, can also be important. Earlier we saw the example of the OCI. Its executive director Chris Aniszczyk told Gordon that "I think the industry has changed over the years. Open source is more prevalent. People have learned a lot of lessons around lock-in, and they don't want to repeat the mistakes. The visualization fiasco with the format, VM, all that, that's a painful memory in a lot of people. People are worried about paying the 'VMware tax.' Lots of lessons have been learned."

Portability is closely tied to, and in many ways a product of, aspects of openness such as this. Without being able to deploy on a choice of infrastructure, you don't have portability. Portability requires thinking about how applications and data can be moved from one place to another and assessing the impact of such a move. Multiple technologies can come into play, although, ultimately, it's about making business decisions regarding the degree to which you're tied or not tied to a specific vendor or provider in some manner.

At the same time, there's a general recognition that you need to choose when and where the time and place are right for standardization and when it makes sense to let approaches compete or details to sort themselves out. This is the messy bazaar aspect of open source (to use Eric Raymond's Bazaar vs. Cathedral metaphor).

Clouds are the ultimate cathedrals. They contain. They prescribe. They package. They're the ultimate bundle.

Preserving the freedom to tinker.

Open source, by contrast, is the ultimate force for unbundling. Mix and match. Modify. Tinker. Move.

To be clear, it doesn't need to be either/or. Public cloud infrastructure and software providers depend heavily on open source software and are active to greater or lesser degrees in a variety of important open source projects. Open source applications can be written so that they are portable across many cloud platforms. Clouds don't need to be treated as the vertical stacks that were once simply the-way-systems-were-built, a model that largely gave way to horizontal layers such as microprocessors, operating systems, and databases developed by different specialist vendors and brought together at the end user. (Which often became just a different source of lock-in. The new boss same as the old boss and all that.)

Open source changed this. It redefined the economics of IT and gave control over their software back to users. It made possible a style of community-led development that had only been possible in very limited ways previously. It effectively turned what had been a top-down vendor-led approach to designing and delivering product into one that springs from ideas coming from everywhere. Open source development can look messy compared to integrated proprietary products, but time and time again, the choice, flexibility, and innovation stemming from open source have won out.

But the current era also values packaging and experience more than in the past. And that's the challenge that open source adherents must collectively address.

Simplicity is a challenge because it runs counter to developer and, especially open source developer, instincts to offer more choices, more options. It runs counter to a preference to let users select among alternatives in a sort of Darwinian free-for-all. How many desktop environments are available for Linux again?

Simplicity doesn't come naturally to open source. There's usually no central authority carving out unnecessary features and holding firm to a streamlined architectural vision.

Simplicity isn't inherent in all cloud options either. At this point navigating the Amazon Web Services catalog of services is a daunting task. But, to the degree that open source software projects can simplify installation, simplify configuration, and simplify ongoing operations, they'll see even more adoption. Containers and automation tools such as Ansible are making great strides to abstract away a lot of the complexity around software provisioning and configuration.

Integration has been one of the biggest challenges to adopting open source software over time. Tight integration would seem to fly in the face of an ethos of independence from specific technology tracks and specific vendors.

But it's not a binary choice.

Consider the technological innovation happening around containers and DevOps. On the one hand, this creates enormous possibilities for new types of applications running on a dynamic and flexible platform. And this continues to happen. But it doesn't preclude *also* having an integrated (but extensible) container platform.

And for many organizations, channeling and packaging the rapid change happening across a plethora of open source projects isn't easy—and can end up being a distraction from the ultimate business goals. With container formats, runtimes, and orchestration increasingly standardized through the OCI and CNCF (where Kubernetes is hosted), there's increasing interest from many ops teams in deploying a tested and integrated bundle of these technologies.

Based on a series of interviews, market researchers IDC found that:

> IT organizations that want to decouple application dependencies from the underlying infrastructure are adopting container technology as a way to migrate and deploy applications across multiple cloud environments and datacenter footprints. OpenShift provides a consistent application development and deployment platform, regardless of the underlying infrastructure, and provides operations teams with a scalable, secure, and enterprise-grade application platform and unified container and cloud management capabilities.[54]

If you consider the differences between perceptions about open source as it was starting to become important to businesses and today, one of the big changes is **confidence** around open source security, support, and reliability. Much of this was, in fact, largely present early on but it took a while to get the word out. The confidence provided by enterprise open source packaging is one aspect that has led to the shift in perception.

This shift comes from how the open development model allows entire industries to agree on standards and encourages their brightest developers to continually test and improve technology. Developing software in collaboration with users from a range of industries, including government and financial services, provides valuable feedback that guides security-related discussions and product feature implementations. Collaborating with communities to solve problems is the future.

[54] https://www.openshift.com/sites/default/files/idc-business-value-of-openshift.pdf

This collaboration brings additional benefits. As Paul Cormier, the president of Products and Technologies at Red Hat, wrote recently:[55]

> This commitment to contribution translates to knowledge, leadership, and influence in the communities we participate in. This then translates directly to the value we are able to provide to customers. When customers encounter a critical issue, we are as likely as anyone to employ the developers who can fix it. When customers request new features or identify new use cases, we work with the relevant communities to drive and champion those requests. When customers or partners want to become contributors themselves, we even encourage and help guide their contributions.

Open source software suppliers also put a wide range of processes and services in place to further enhance confidence in open source software. Modern security means shifting from a strategy that is built around minimizing change to one that is optimized for change.

Enterprise open source software also requires code review and testing methodologies, a supply chain that's secured by digitally signing all released packages and distributing them through secure channels, and a dedicated Product Security team (such as we maintain at Red Hat) that analyzes threats and vulnerabilities against all our products every day and provides relevant advice and updates.

Finally, **Experience** is where the rubber hits the road. Everything comes down to delivering an experience through the software and the way in which it is packaged.

[55] https://www.redhat.com/en/about/blog/what-makes-us-red-hat

Open source brings freedom. Open source brings flexibility. Open source brings choice. Open source brings independence.

But open source also must keep its eye on delivering those attributes to users with the minimum of friction. This means moving beyond thinking about software in the traditional sense—and instead enabling the streamlined delivery of digital services.

The innovation taking place in cloud-native development today provides many options to make this approach a reality. And the open source development model has proven to be hugely successful. But there remains the need to focus on and embrace packaging principles to deliver a simplified and enhanced user experience. A better experience.

Not everything about computers and software these days is related to packaging, but a lot is. How can users acquire software faster and run it more easily? How should software be packaged up so that we can run it wherever we like? How do we resolve the tensions between the siren song of public cloud convenience and open source freedom and flexibility?

In this book, Red Hat's Gordon Haff and William Henry take you on a journey through the history of packaging, pointing out along the way the clear parallels to how software packaging has evolved and continues to evolve. The retail industry and other packaging pioneers have had many lessons to teach and much to say about how we should think about software packaging. Today, open source software innovations in automation, container platforms, and assured software supply chains increasingly support the user directly, rather than just serving as a way to box up some bits.

They simplify, integrate, and improve the overall software experience in the way that consumers accustomed to smartphones and on-demand content expect.