ADVANCED SYSTEM ANALYSIS AND DESIGN

HOT TOPIC REPORT

THE SOFTWARE REVOLUTION BEHIND LINKEDIN'S GUSHING PROFIT

PALAK TATER

U62790011

Introduction:

LinkedIn is a business and employment-oriented service that operates via websites and mobile apps. Founded on December 28, 2002 and launched on May 5, 2003 it is mainly used for professional networking, including employers posting jobs and job seekers posting their CVs.

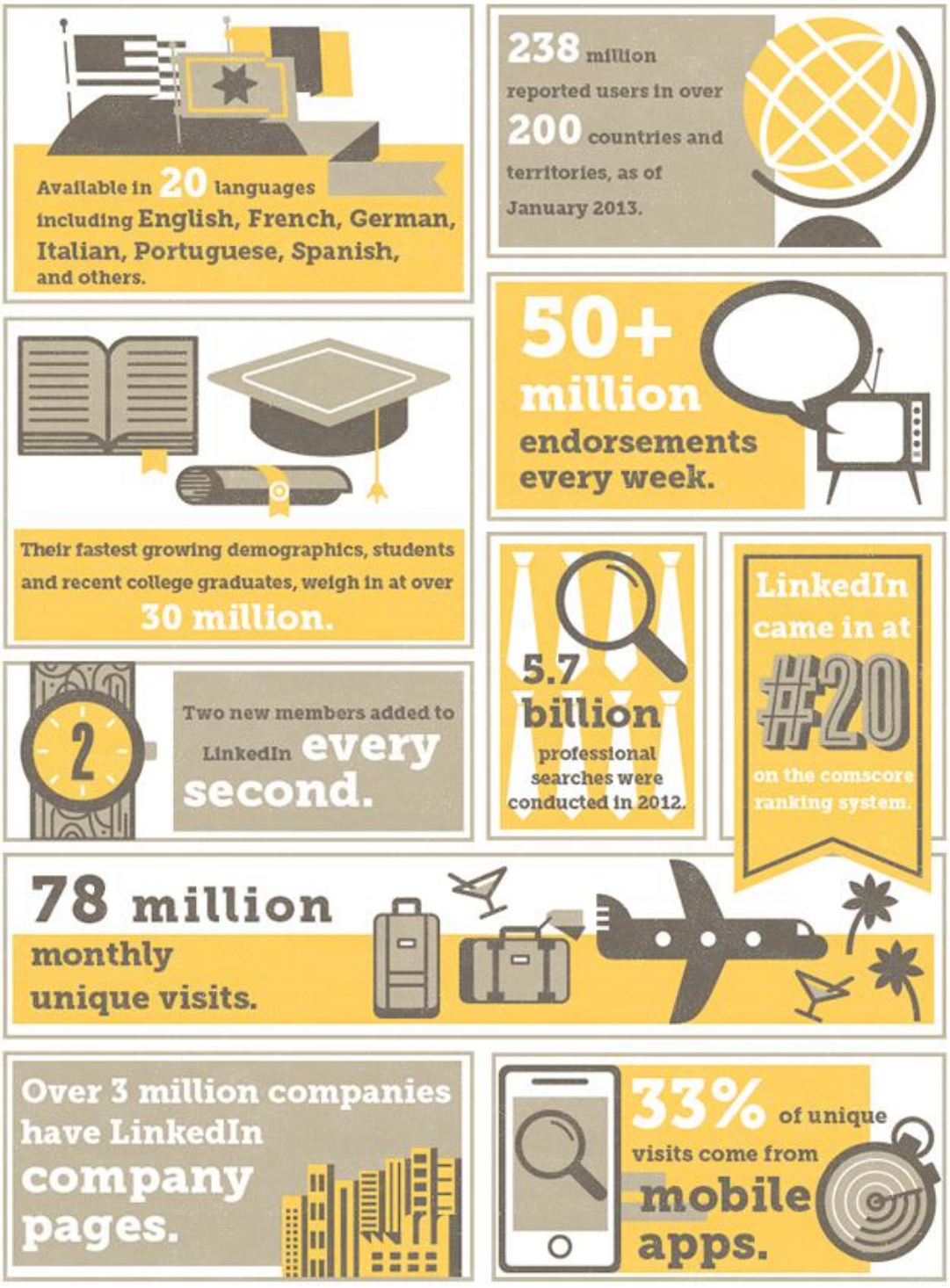
LinkedIn is a Wall Street darling, its stock up more than threefold in two years on soaring revenue, spiking profits, and seven straight quarters beating bankers' estimates. But LinkedIn's success isn't just about numbers: an impressive acceleration of LinkedIn's product cycle and a corresponding revolution in **how LinkedIn writes software is a huge component in the company's winning streak.**

Much of LinkedIn's success can be traced to changes made by Kevin Scott, the senior vice president of engineering and longtime Google veteran lured to LinkedIn in Feb. 2011, just before the buttoned-down social network went public. It was Scott and his team of programmers who completely overhauled how LinkedIn develops and ships new updates to its website and apps, taking a system that required a full month to release new features and turning it into one that pushes out updates multiple times per day.

"*Without having done all this work to change how we built our software," says Scott, President of LinkedIn, "it literally would be impossible for us to be building endorsements, and the new influencers product, and the new version of profile and the things that are happening in mobile, and the upgrades we made to the recruiter product, and all of these dozens of significant changes that are rolling out."*

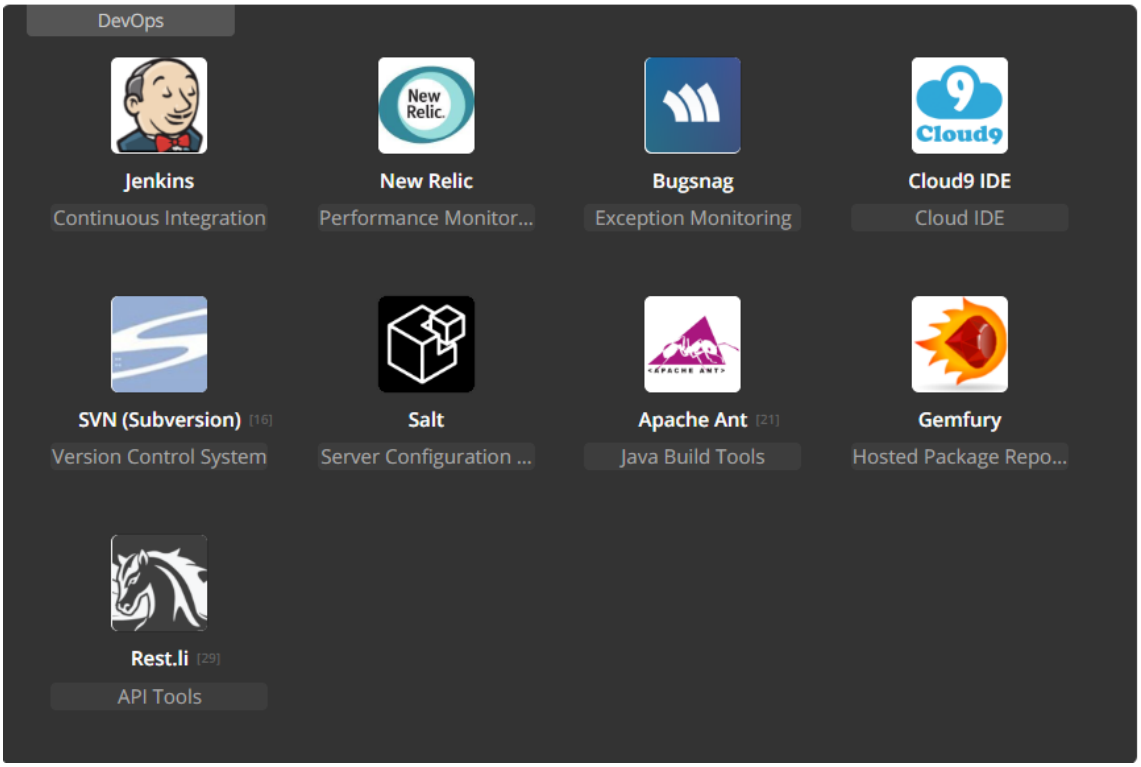
The Software Revolution behind LinkedIn's gushing profit

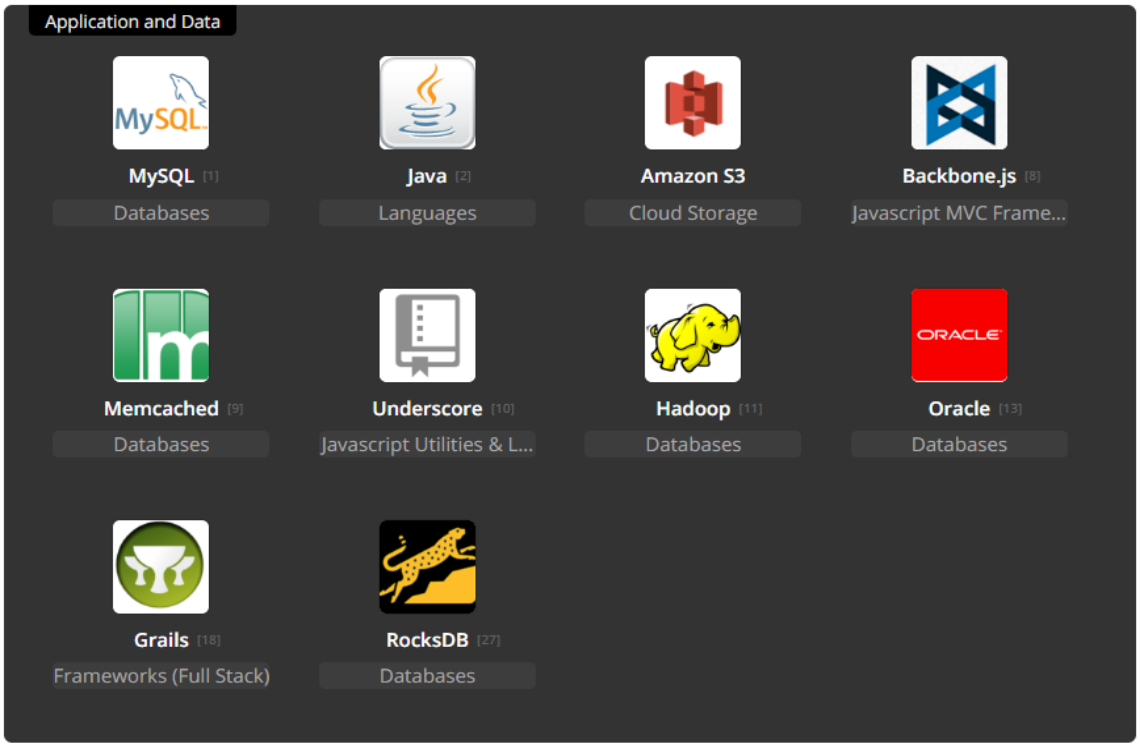
Key stats and demographics



The Software Revolution behind LinkedIn's gushing profit

LinkedIn’s Secret Sauce – The Technology Stack





Main technologies and principles of LinkedIn Scalability: Kafka, Expresso, Hadoop, Voldemort and Azkaban, Oracle, MySQL, Galene, REST.LI API which provides data consistency across the enterprise. These are supplemented with multi-level caching, CDNs and distributed data stores for speed.

Software Development Technique Behind LinkedIn’s gushing profit

LinkedIn's newly-adopted software development methodology is known as "continuous deployment." Under continuous deployment, a developer writes new code in tidy, discrete little chunks and quickly checks each chunk into the main line of software shared amongst all LinkedIn developers, a line known as “trunk” within the software version control systems like GitHub standard in the tech industry. Newly-added code is subjected to an elaborate series of automated tests designed to weed out any bugs. Once the code passes the tests it is merged

The Software Revolution behind LinkedIn’s gushing profit

into trunk and cataloged in a system that shows managers what features are ready to go live on the site or in new versions of LinkedIn's apps.

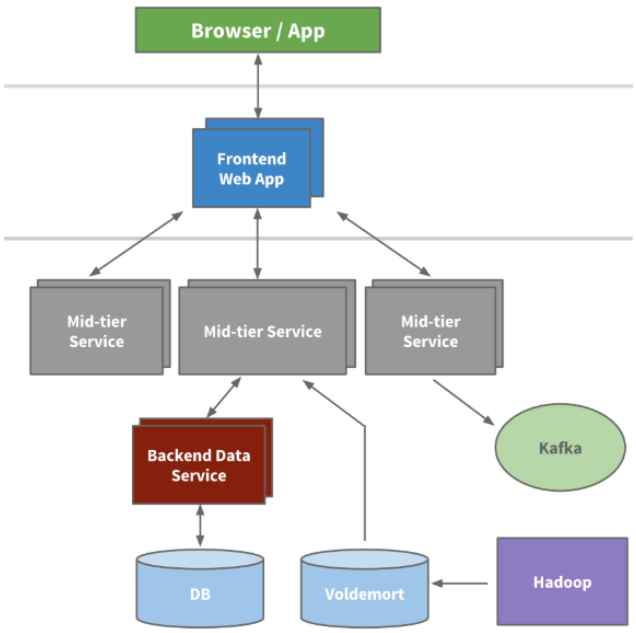
LinkedIn's prior system of software development was more traditional, involving software "branches" forked off from trunk and developed in parallel over a period of weeks or days. A developer would finish a big batch of code corresponding to some feature and then lobby for this feature branch to be merged into trunk. Once merged into trunk, the feature would again need to be tested to ensure it did not break any of the other new code checked into trunk at the same time. Bugs and outright broken software are common under this so-called "feature branch" system, since typically several big batches of code, each written in isolation by a separate team, are merged into trunk at once. To avoid such meltdowns, managers tended to tightly limit the number and scope of new features mashed together each month, slowing down a company's development cycle. LinkedIn is hardly the only company to use continuous deployment. Scott had experience with the system from prior gigs, and other internet companies have embraced the practice as well, including handcrafted-goods marketplace Etsy and Facebook. But LinkedIn's big switch to continuous deployment has been linked to very concrete and visible financial success, helping lend credence to the practice and potentially helping to accelerate the delivery of software across the tech industry.

"We had to go from this model where developers were developing their code in relative isolation and then raising their hand and say, 'please integrate my feature branch into the release branch and test it and qualify it for me and push it out to the site in the appropriate release window,'" Scott says. "We wanted to be at the point where... as soon as they were checking in their code... it was qualified and releasable... that anything sitting in trunk must be releasable at any point in time, and if it's not releasable it's just as significant as a site emergency. Stop all forward software development and everyone is all hands-on deck to get trunk fixed."

A Brief History of Scaling LinkedIn Software

The early years

LinkedIn started as many sites start today, as a single monolithic application doing it all. That single app was called LEO. It hosted web servlets for all the various pages, handled business logic, and connected to a handful of LinkedIn databases. As the site began to see more and more traffic, their single monolithic app Leo was often going down in production, it was difficult to troubleshoot and recover, and difficult to release new code. High availability is critical to LinkedIn. It was clear that they needed to "Kill Leo" and break it up into many small functional and stateless service.



An example multi-tier service-oriented architecture within LinkedIn

The Modern Years

To collect its growing amount of data, LinkedIn developed many custom data pipelines for streaming and queueing data. For example, they needed the data to flow into data warehouse, and needed to send batches of data into their Hadoop workflow for analytics, they collected and aggregated logs from every service, collected tracking events like pageviews, needed queueing for our in Mail messaging system, and they needed to keep their people search system up to date whenever someone updated their profile.

As the site grew, more of these custom pipelines emerged. As the site needed to scale, each individual pipeline needed to scale. Something had to give. The result was the development of Kafka - Linkedin's distributed pub-sub messaging platform.

Kafka became a universal pipeline, built around the concept of a commit log, and was built with speed and scalability in mind. It enabled near real time access to any data source, empowered our Hadoop jobs, allowed us to build real time analytics, vastly improved LinkedIn's site monitoring and alerting capability, and enabled them to visualize and track all call graphs. Today, Kafka handles well over 500 billion events per day.

LinkedIn today is still mainly a Java shop, but also has many clients utilizing Python, Ruby, Node.js, and C++ both developed in house as well as from tech stacks of our acquisitions. Moving away from RPC also freed them from high coupling with presentation tiers and many backwards compatibility problems.

What is KAFKA?

Apache Kafka is a publish/subscribe messaging system with a twist: it combines queuing with message retention on disk. Think of it as a commit log that is distributed over several systems in a cluster. Messages are organized into topics and partitions, and each topic can support multiple

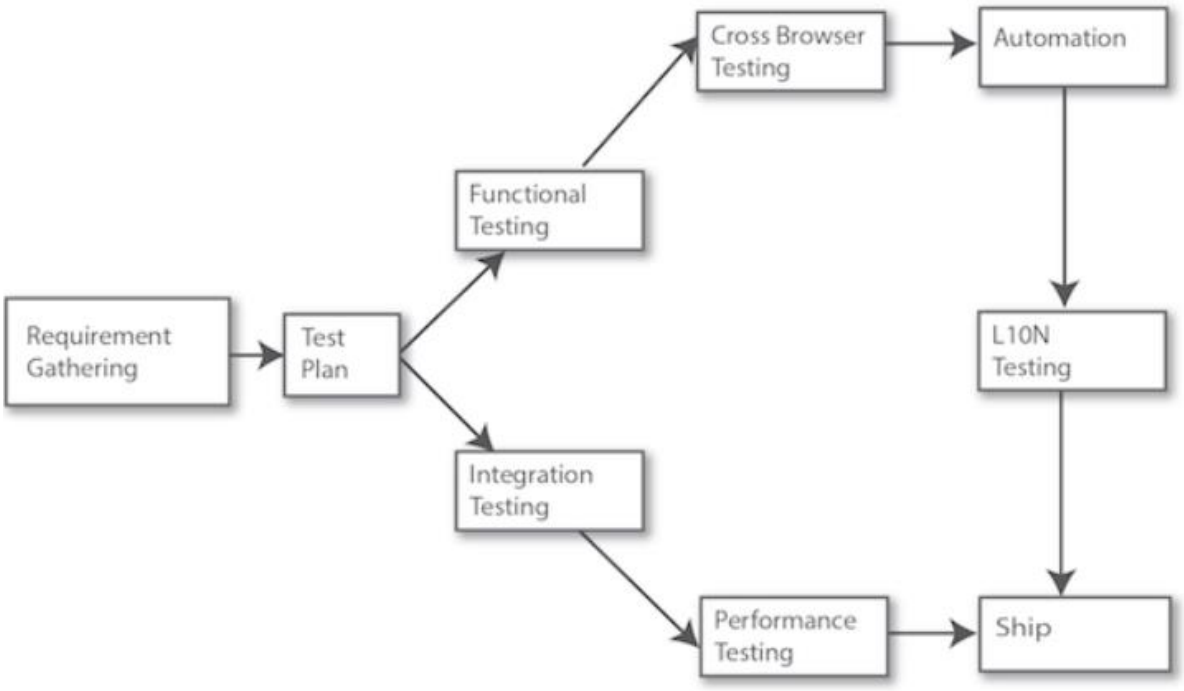
publishers (producers) and multiple subscribers (consumers). Messages are retained by the Kafka cluster in a well-defined manner for each topic:

- For a specific amount of time (measured in days at LinkedIn)
- For a specific total size of messages in a partition
- Based on a key in the message, storing only the most recent message per key

Quality Control - LinkedIn's Testing Methodology

A typical test team at LinkedIn includes product managers, development engineers, software engineers in test, and quality engineers. They have a dedicated team of engineers that rotate weekly and whose only job that week is to fix and close bugs.

Test Strategy and Process



Test Strategy

A testing lifecycle can run multiple weeks long and it begins by spending the first week preparing for the official kick-off. Quality engineers do a product spec review and create a test plan along with an estimate of the release date. The next few weeks are spent in writing detailed test cases in an internal tool called Test Manager. Training sessions are held in parallel to educate product development engineers on the testing lifecycle. Once test cases and training is complete, the testing begins.

Tasks are assigned daily. A task dashboard tracks progress and monitors newly filed tickets, while daily stand-ups are used to discuss issues. This aggressive status-tracking approach helps them stay current and continuously plan towards the release.

Integration Testing

Integration testing and functional testing run in parallel. SETs (Software Engineers in Test) perform tests on APIs for positive, negative, and boundary conditions. Integration tests are written in TestNG - a Java testing framework. These tests run every night and on every code check-in. (Details on Integration testing will follow in subsequent posts.)

Performance Testing

Performance testing is an important part of LinkedIn's release cycle. SETs start with backend load testing – LinkedIn runs load tests and get performance metrics, such as peak QPS values and response times. They then use Apache JMeter and other internal tools to ensure that our system can handle production load per business criteria. At this point, engineers start frontend performance testing and produce metrics such as page load, JavaScript execution time, page size, and page component load time. Bugs are filed and fixed, and they re-run tests until they reach our performance goals.

Functional Testing

The test team, mostly quality engineers and a few development engineers, are engaged in executing test cases manually as part of functional testing. Features are distributed among testers, taking particular care not to have the same engineer develop and test a feature. Bugs are filed and aggressively addressed to ensure a shorter bug lifecycle.

At LinkedIn they developed LiX, an online experimentation platform and manages the lifecycle of all tests and experiments. Every product feature is behind a LiX test and functional testing is performed with LiX turned on and off. The test team is involved in testing the tracking of pagekeys, tracking codes on Kafka consumer, device testing on iPad and iPhone, and finding gaps in metrics testing.

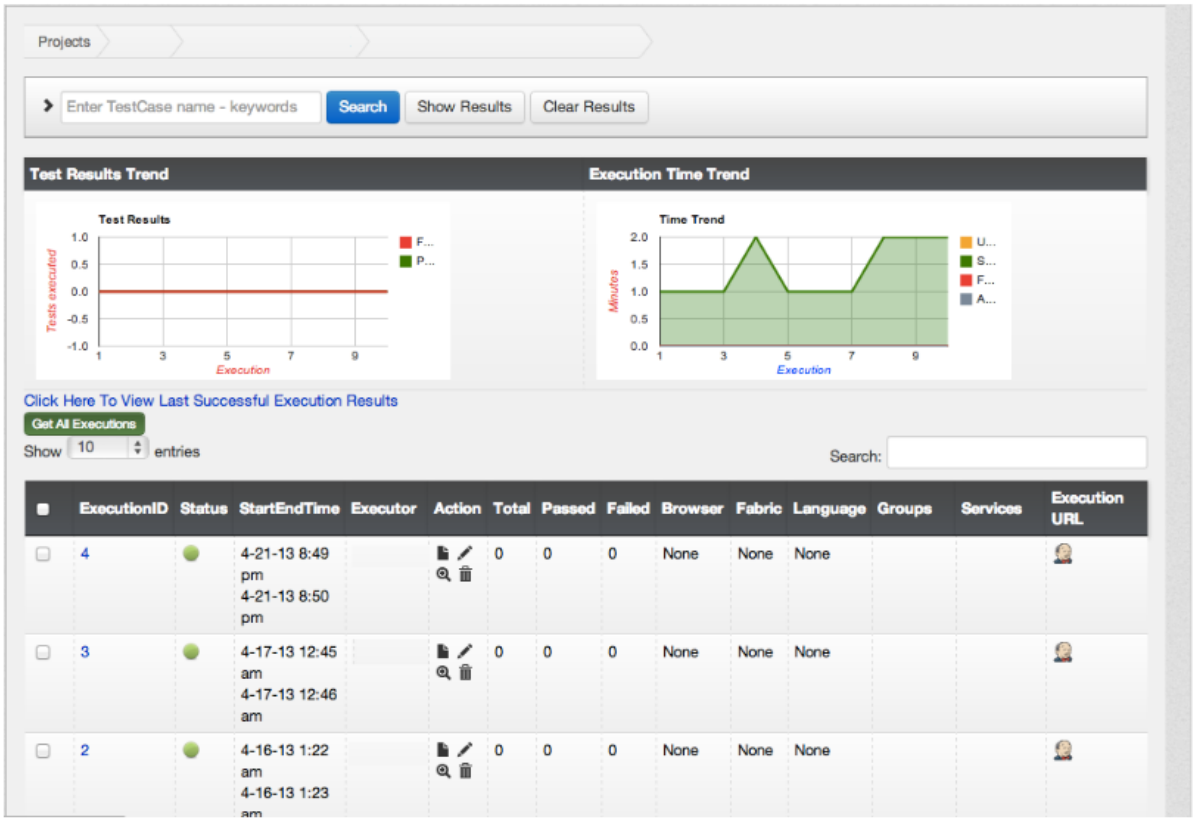
Cross-Browser Testing

After functional testing is complete, the team transitions to browser testing, which takes place across different versions of IE, Firefox, Safari, and Chrome. It's important to find and fix these browser issues early and collaboration with the web dev team is particularly important here.

Automation

Automation begins next with each tester writing scripts for their assigned feature, using Ruby and Selenium as the automation tools. Scripts are i18n (internationalization) compatible and are passed on to the i18n teams to begin their test cycle. The test cases are marked as automated and tracked on Test Manager.

The product manager and offshore resources are involved in bug verification. They monitor bug activity using the dashboard, tracking the number of defects captured, and fixed daily. This helps them review any reopened bugs and ensure there are no blockers close to the release date.



Test Plan Execution on Test Manager

After all the automation scripts are ready, regression tests are run every day on all supported browsers through Test Manager against new builds, to capture bugs as they appear. A bug bash is organized to get feedback from different teams, and bug triggered automation is done as part of bug verification.

L10N Testing

After automation is complete, test scripts are passed on to the L10N (localization) team to execute in different languages. The team adds code to take screenshots in each language, which are then used by translators to make sure the UI elements make sense contextually. Any bugs filed are immediately addressed by the bug fixers. Only after the L10N team signs off on the UI, they roll out the product in various languages.

Ship

When code is pushed to production, the test team runs production sanity checks to ensure that functionality is not broken. LiX is slowly ramped to 100%, and we constantly monitor and logs in production for any anomalies.

At LinkedIn, they set and strive towards the highest quality in our products. The test team, working with our development engineers, act as gatekeepers to guarantee quality releases.

Seven fundamental dimensions of Software Craftsmanship

Code Quality

It all starts with the code LinkedIn writes. One should always assume that in the future, someone else will take over the code. One needs to put oneself in his/her own shoes, asking if the code is clear and modular enough that one'll be able to understand what the code is trying to do easily. An organization should make this easier by providing clear code quality guidelines for every language that is supported. These can make writing quality code standard practice.

Users don't see it, but it should still look like one of the most beautiful and elegant things you've seen, because that makes it easier to be maintained and expanded upon.

Scalability

Scalability can be thought of as the ability for a software system to remain functional when the load spikes. If the load on the system spikes the day after a holiday break (which is usually the case at LinkedIn), the products and services should be able to handle that. Building extensible and scalable systems has always been a primary concern for LinkedIn, which is why they have continuously invested in building massively scalable systems and components such as Kafka, a high throughput messaging system that allows us to route more than 800 billion messages per day at LinkedIn.

As LinkedIn grew up in popularity, so did the needs of its members and customers to analyze data at scale and in real time, such as the needs to understand how a LinkedIn ad campaign or a given set of job postings perform, based on hundreds of profile dimensions, such as Industry, Function, or Geography. It needed to do rollups and drill-downs across billions of rows and hundreds of dimensions. Instead of building a custom ad hoc solution, LinkedIn decided to build an online analytics service to serve the needs of the wide range of products and services that LinkedIn offers. This is how Pinot, a real-time distributed OLAP datastore, was born. The ROI has been phenomenal as Pinot now supports all online analytics capabilities across LinkedIn's products and services.

Building reusable components that will be leveraged by other use cases also contributes to scalability.

High Availability

LinkedIn's systems cannot go down. Ever. They have customers all over the world who rely on our service being available 24/7. If a single failure causes service interruption, they've failed. Losing a node should not cause a cluster to fail, and catastrophic failure like data center level error conditions should not cause an interruption of service either. High availability is such an important concern for LinkedIn that they have built Helix, a cluster management framework with high-availability built in. Helix automates reassignment of resources in the event of a node failure, a node recovery, or when expanding or reconfiguring such a cluster, thereby preventing interruptions of service when specific parts of the system are in error condition or are being worked on for maintenance.

Security

The Internet is a very unsafe place, and we owe it to our users to make our products and services as secure as possible. We all need to write software that does not introduce security vulnerabilities or jeopardize our users' data. Correcting potential vulnerabilities early in the software development lifecycle significantly reduces risk and is way more cost effective than thinking about security after the fact and having to release frequent patches. Security needs to be built in. LinkedIn incorporates security requirements in the code they write, however this information cannot be revealed due to confidential reasons.

Simplicity

Just like one should assume that their code will be maintained and extended by another engineer in the future, the systems we design should be as simple as possible for the task at hand. After all, other folks will have to maintain and operate it. There's a quote by Albert Einstein that I like a lot: "You should make things as simple as possible, but not simpler."

If we make things more complicated than they need to be by over-engineering systems, we create friction and impair the team's ability to iterate on and enhance the software. This is a

balancing act, for sure since we need to build software to solve the complex problems at hand. But build in a sub-optimally simple fashion and you introduces downstream friction for folks who operate it and, ultimately, for your users. One of the reasons LinkedIn has been able to continuously enhance some of the most complex systems we've built at LinkedIn, such as the infrastructure they use to run machine learning algorithms at scale, or our auction-based ad system serving millions of queries per second, is our continued focus on making these systems as simple as possible.

Performance

A slow page is a useless page. At LinkedIn, we've proven through experimentation that degradations in performance have serious impact on our member experience, the health of our ecosystem, and also can hurt top key metrics such as signups, engagement, or revenue (not that we were expecting different findings, but we like to test everything through our experimentation platform to know the magnitude of the effects). Every single page or application should be instrumented for performance measurement. Tracking and fixing performance degradations as we introduce more features, products, and services is a very healthy habit. Every engineer must own performance as a key aspect of the code they write. This is why leveraging their RUM (Real time User Monitoring) framework, every engineer at LinkedIn can (must) instrument the pages they build for real time performance tracking. Aggregate page latencies are automatically displayed in a user-friendly dashboard that allows you to see how fast a given page or application is loaded and rendered, and to slice, dice, and drill through performance measurements across devices (web, iOS, Android, …) or geography.

Operability

Operability is the ability to keep a software system in a safe and reliable functioning condition. We have several ways of instrumenting this at LinkedIn. One way is by measuring the number of

Site Reliability Engineers (SREs) that are needed to operate our software on a node basis. Is it one, 10, or 100 SREs for a thousand nodes?

Other metrics that LinkedIn uses include: How long does it take to bring an instance to life? If a service takes 15 minutes to boot, that is not very operable. If a service generates millions of meaningless exceptions per hour and clogs the logs, that's not very operable either. To enhance operability, the best folks to engage with are the folks who operate the systems every day and every night.

Whether you're a full Devops shop, or whether you partner with an SRE or sysadmin team to operate your software, it's always a good practice to ask operationally-minded folks what to do to make a given service more operable. The benefits are numerous, including: (1) it enhances your ability to recover from an error condition if your systems are optimally operable, and (2) folks operating the systems can spend more of their valuable time building new things that create further leverage as opposed to spending time trying to keep the system alive and serving.

How SCALA and CONTAINERS hugely impact LinkedIn's Software Methodology

Chris Conrad, Engineering Manager, a part of the Search, Network and Analytics team at LinkedIn presented Norbert at ScalaDays . Chris works on the LinkedIn Social Graph represented by some 65+ million nodes, 680+ million edges and 250+ million request per day.

He explains that their People Search Engine receives around 15 million queries a day, or 250 queries per second with up to 100 tokens per query. Queries are satisfied by a scatter-gather approach across a large server farm which presented a challenge for efficient message routing and resource management, reliable 24x7 operation and easy application development.

Norbert is a framework written in Scala that makes it fast and easy to write asynchronous, cluster aware, message-based client/server applications. Built on Apache Zookeeper and JBoss Netty,

Norbert provides out of the box support for notifications of cluster topology changes, application specific routing and load balancing, scatter/gather style APIs and partitioned workloads. Written by the SNA team at LinkedIn, Norbert is open sourced under the Apache License.

Chris explained that they use Scala because it makes concurrent support easier with Actors and promotes code reuse with traits and mixins. He found it nice that the "cake pattern" could be implemented elegantly using self-types to declare dependencies and have the compiler do the checking. He says that the "killer feature [of Scala] is the seamless integration of Java and Scala, making it low risk to introduce, and reduces the overhead to experiment. It made software development more fun and a lot less frustrating.

Project Inversion was an effort to decentralize the services that make up the application and rethink the development of new code. As part of Project Inversion, LinkedIn pivoted to a fine-grained microservices-based approach. Each of the thousand or so services that make up the app were managed independently with an owner and development team that released new features when they were ready.

It’s not an easy environment to manage. Many services depend on each other, so when one is updated another must be too; sometimes dozens of services need to be updated simultaneously. It does become complicated to choreograph that whole effort. If too many services rely on one another that could be a sign the boundaries of services need to be redefined, though.

The company launched its private cloud called LinkedIn Platform as a Service (LPS), which is used for managing new application development and automating the underlying hardware needed run those apps. With LPS, the entire infrastructure was pooled together and automatically allocated to the services that need it.

There are two major custom developed components of LPS. One is named Rain, which is an API-driven infrastructure automation platform. Developers write their code and through APIs request the amount of memory and CPU needed, which Rain configures. With Rain, applications no longer need to ask for an entire machine as a unit of resource, but instead can request a specific amount of system resources.

A second major component (there are many others) of LPS is named Maestro, the conductor of LPS. Maestro is a higher-level automation platform compared to the infrastructure-focused Rain. Maestro handles the process of adding new services into the application, configuring them with other services, routing traffic to the necessary locations, registering the feature with the broader system and basically making sure new services fit nicely into the broader environment.

Both **Rain and Maestro** were designed to manage code developed and packaged in application containers, specifically using the Docker runtime. LinkedIn runs LPS on a bare-metal, non-virtualized environment with containers. There's no overhead of a hypervisor, multiple operating system instances and virtual machines.

Containers also allow LinkedIn's engineering team to enforce fine-grained security controls. The goal is to give each service the impression that it is running on an empty host with nothing else on the machine. If a bug or a hack were to compromise a component, the idea is the blast radius of the incident would be limited. Application containers help do this: Configuring name spaces of containers in the Linux kernel allows for the container processes to be hidden from other containers on the same host. Each container has its own network name space and IP address too.

The Code Review process at LinkedIn

Usually, a developer, for example, is working on a feature and that feature requires them to make a change to a service or a part of the code base. Say the developer wants to make a change in the Android application; LinkedIn team goes ahead, write the tests, write the change to meet those tests to make sure that the change being made is well tested. And then they

create a code review submission. They use a tool called **Review Board**. They also have a little command line tool that lets them create a code review from their local git repository, Git workspace.

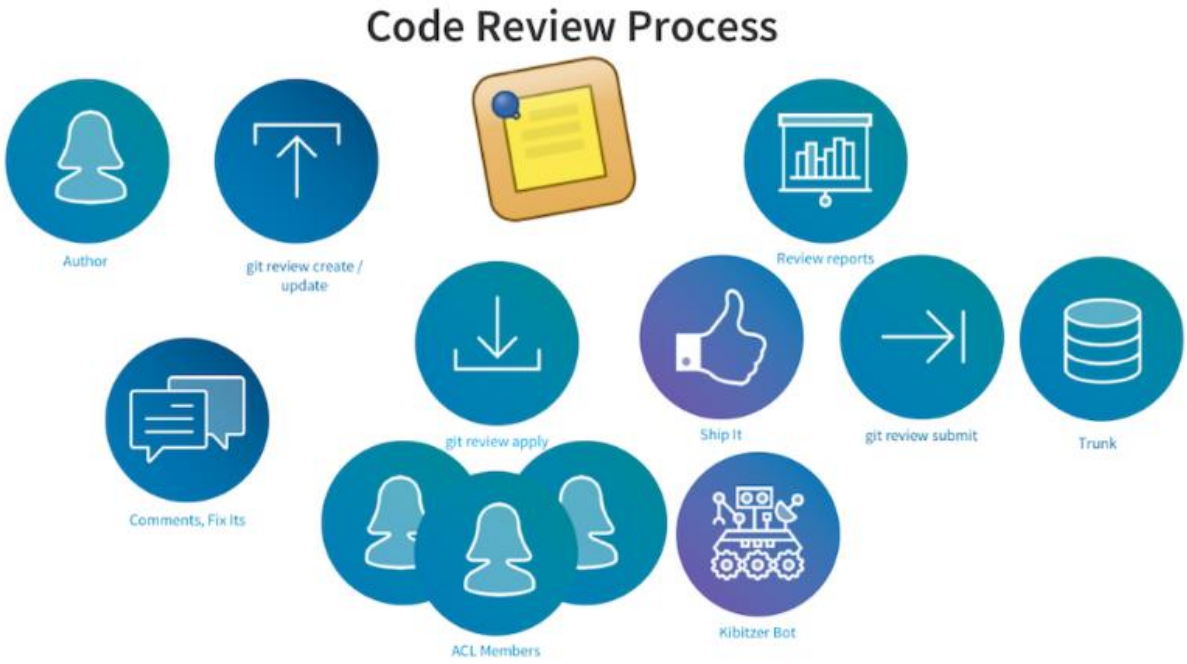
That creates a Review Board ticket and notifies people who are responsible for maintaining the part of the code base I'm changing. Those authors then look at Review Board, look at the changes to the code change, and possibly offer suggestions. This might happen in a single iteration. Let's say we got everything perfect and it just looks great and the maintainers of the code base look at it and say, "Looks great, thank you very much," they would then give it a sign-off for a contribution to the code base.

We would then use another command line tool to submit the change. It goes through a battery of automated tests, then it's committed to the code base and it's ready for their continuous deployment pipeline to build it and make it deployable for the next release of their Android application.

Software Code Review at LinkedIn

Ownership is a key aspect of large-scale software development. The owner of source code usually refers to the person that invented the code. However, larger code artifacts, such as files, are usually composed by multiple engineers contributing to the entity over time through a series of changes. Frequently, the person with the highest contribution, e.g. The most number of code changes, is defined as the code owner and takes responsibility for it. Thus, code ownership relates to the knowledge engineers have about code. Lacking responsibility and knowledge about code can reduce code quality.

Modern code review in Open-Source Software



Any contributor at LinkedIn can submit a contribution for review to any repository at LinkedIn. The code review process we have lets us scale code ownership to the organization level. The tooling helps you find the right owners who would need to review your contribution; it also lets you know what Slack channel you can find them in. Since we know that context is important when doing a code review, you can choose to apply the submitted changes locally and review them in the IDE. For Java, we favor IntelliJ at LinkedIn. The conversation about the code happens in Review Board, on Slack, or sometimes face-to-face. It's preferable to use a high-bandwidth channel when there is major confusion around an issue. Once all issues have been addressed, and a "Ship It" has been given out, tools help us add review sign off information to the commit message and off it goes to trunk and into our continuous deployment pipeline.

Setting up a code review practice of your own

If you are ready to setup a code review practice in your organization to help scale collective code ownership, there are somethings that you might want to consider.

Code reviews can take between 6-10 hours a week according to surveys across industry and open source so make sure that engineers have the time to conduct thorough code reviews. Code review effectiveness drops when reviewers rush the review process. Implementing a standardized code review system is a culture change, not just a process change. Asking engineers to submit their work for review might not be as simple as you think, especially for engineers who have spent most of their professional careers without code reviews. You might need to consider changes to training and onboarding, and potentially even how you go about hiring and conducting performance evaluation. At LinkedIn, code review participation is mandatory, and participation , as both an author and as a reviewer, is taken into consideration when performing promotion evaluations.

Code reviews are performed by people who have to balance multiple priorities, and there can be a natural tension between people submitting code for review and people doing the code reviews. This is especially the case when each side's priority is different; the submitter may be focused on delivering a product change quickly, while the reviewer could be more concerned with the long-term quality and health of the software product being modified.

For these reasons, it's important to instrument the review process and monitor it for trouble spots and bottlenecks. You may find under-resourced teams, design and architecture decisions that are creating bottlenecks, or even interpersonal communication and collaboration issues. Since most of the code review feedback is not about defects or defect removal, code reviews are not a substitute for other QA processes, like testing.

Tips for code maintainers

If you already have a code review process in place and you want to focus on improving as a reviewer, you might want to try making the following adjustments.

Take your time

Don't do it all at once. Your ability to do a good job reviewing code diminishes with the amount of time you spend on a review. In "Best kept secrets of peer code review," Cohen suggests that the total time spent on any given review needs to be around 60 minutes and should not exceed 90 minutes.

Focus on useful feedback. Remember that you are a maintainer, so you can always clean up the code later if needed—the contributors are trying to add value to your system with this change.

Tips for contributors

If you already have a code review process in place and you want to improve as a contributor, you might want to try making the following adjustments. Make your changes small. The number-one issue faced by maintainers is large changes. In "Best kept secrets of peer code review," the author recommends that changes should be kept to around 200 LOC and not exceed 400 LOC.

Respond to feedback promptly. Act on feedback from the review in a timely manner, because your reviewers need to manage the time they spend on code reviews with the other priorities they have. **Find the right reviewer**. For a code review, someone who's familiar with the particular code will be able to give you the most useful feedback. Communicate with compassion. Assume the reviewer has the best intent and is trying hard to help you make this contribution.

Conclusion:

Recommendations to the Manager:

- One of the biggest benefits of implementing company-wide code reviews has been increased standardization in our development workflow. Every team at LinkedIn uses the same tools and process for doing code reviews, which means that anyone can help with reviewing or contributing code for another team's project. This eliminates problems like "I could fix the bug in their code, but how would I build that code and submit the fix?" This, in turn, helps increase collaboration across different teams in the engineering organization.
- Plant your code and let it grow - LinkedIn loves open source. This mentality—opening everything that could be useful within and beyond LinkedIn, and of doing the "dirt work" necessary to make it useful to external developers—is what makes LinkedIn such an impressive organization to watch. Not only does this result in better software for LinkedIn and everyone else, it also helps LinkedIn recruit the best engineers. As Qin says, "Many companies claim to build great software, but the best way to prove it is to open-source it. If you open-source something, that means you're proud of it, and it's useful not just for you but for others as well." Decades after open source launched in earnest, too many corporations mistakenly view open source to source software for zero dollars, rather than to change how they build software and do business. But such companies will be left behind by LinkedIn, Facebook, and others that take an open-source-first approach to code and are willing to invest the labor to make it pay off.
- Apache Kafka was originally developed by LinkedIn and was subsequently open sourced in early 2011. Apache Kafka is based on the commit log, and it allows users to subscribe to it and publish data to any number of systems or real-time applications. Kafka® is used for building real-time data pipelines and streaming apps. It is horizontally scalable,

fault-tolerant, wicked fast, and runs in production in thousands of companies. Big MNC's like Apple, Netflix, eBay, Walmart etc. have adopted Apache Kafka.

- In short, continuous delivery is when the default state of your software build is "ready for deployment". As soon as code is written, it is integrated (called continuous integration), tested, built, and configured. The only thing left for developers to do is hit the big red "Deploy" button. Despite speeding up the rate of deployment, continuous delivery helps teams reduce the number of errors that make it into production. Thanks to continuous testing, all errors are caught immediately and sent back to the developer to fix.

In the business world, you're only as good as the technology you use. Tech tools and their rapid evolution have had an increasing impact on the way modern professionals do business and that growth doesn't appear to be slowing down anytime soon. While you may not think your company is really in the tech biz, the path the software industry is taking will still lead you to some valuable destinations.

References:

https://en.wikipedia.org/wiki/LinkedIn

https://www.wired.com/2013/04/linkedin-software-revolution/

https://www.fool.com/investing/general/2015/05/20/this-company-has-the-best-business-model-in-social.aspx

https://jaxenter.com/how-linkedin-switched-to-continuous-deployment-105949.html

https://www.networkworld.com/article/3119832/cloud-computing/case-study-how-linkedin-uses-containers-to-run-its-professional-network.html

https://engineering.linkedin.com/blog/2016/05/from-good-to-world-class--what-makes-software-engineers-excel-at

https://engineering.linkedin.com/blog/2018/09/managing-software-dependency-at-scale

https://stackshare.io/linkedin/linkedin

https://opensource.com/article/17/11/10-open-source-technology-trends-2018

https://oarklimited.wordpress.com/2017/03/21/the-linkedin-technology-stack/

https://www.businessinsider.com/linkedins-weird-names-for-foss-projects-2015-6

https://www.infoworld.com/article/3220667/open-source-tools/how-to-do-open-source-right-linkedin-shows-the-way.html

https://bizzmarkblog.com/4-reasons-backbone-modern-business/