For TripAdvisor, scalability is woven into our organization on many levels - data center, software architecture, development/deployment/operations, and, most importantly, within the culture and organization. It is not enough to have a scalable data center, or a scalable software architecture. The process of designing, coding, testing, and deploying code also needs to be scalable. All of this starts with hiring and a culture and an organization that values and supports a distributed, fast, and effective development and operation of a complex and highly scalable consumer web site.

Stats as of 6/2011

- Over 40M monthly unique visitors (Comscore), 20M members, 45M reviews and opinions
- Over 29 points of sale, 20 languages
- Our mobile offerings are available on iPhone, iPad, Android, Nokia, Palm, and Windows Phone, attracting 10M monthly users
- Over 200M dynamic page requests per day, with all static assets such as js, css, images, video, etc served via a CDN
- Over 2.5B distributed operations (services, database, memcached) are performed each day to satisfy the page requests
- Over 120GB of log files (compressed) are streamed in each day
- 30TB of data on Hadoop, projected to hit 100TB early next year- Daily patches for "need to go out today" features/fixes
- Never, ever, have planned downtime, close to 4 9's of uptime
- Separate deployment in Beijing to support daodao.com
- Weekly release cycle, with daily "patches". Development cycle can be one day, can be one week, can be a month
- Over two dozen small teams (slightly more than 100 engineers) working on over 50 simultaneous projects at a time, with 20-30 of these projects being deployed each week
- Teams include: SEM, SEO, Content management, CRM, Mobile web, mobile apps, social networking (FB), hotels, restaurants, forums, travel lists, video platform, flights, vacation rentals, business listings, content distribution, API, China, APAC, sales and marketing campaigns, data center operations, dev ops, analytics and warehousing, QA

General Architecture

- Open Source Linux, Apache, Tomcat, Java, Postgres, Lucene, Velocity, Memcached, JGroups
- Keep it really simple - easier to build for, debug, deploy, maintain, and operate
- Bank of very simple, stateless web servers, running simple Java and Velocity templates
- Each "functional" area (media, members, reviews, travel lists, etc) is packaged as a service

- Bank of "services" - each service is high level, business oriented API optimized for over the wire performance
- Assume things fail. Either have plenty nodes in a cluster (web servers, service machines), or have true N+1 redundancy (databases and the datacenter itself)

Flow of control

- Flow of control is very simple: request URL's are parsed, content is collected from various services, and then applied to a template.
- Our URL structure has been very well thought out, and has been stable for a very long time, even with site redesigns and code refactorings.
- Requests are routed through the load balancer to a web server. Requests are distributed at random within a cookie based "pool", a collection of servers. Pools are used for deployment management and A/B testing. Requests are essentially stateless - there is "session" information stored in the browser cookie. Logged in users will fetch additional state from the database.
- A Java servlet parses the url and cookie info and determines the content it needs, making calls to the various services. The service API's are defined as Java interfaces, and a servlet will make anywhere from 0 to a dozen service requests. Service calls typically take from 2ms to 15ms. Service calls go through a software load balancer with tuneable retry logic that can be defined on a per method basis.
- Each service has an API that is optimized for the business and/or usage pattern - for example, fetch the reviews for a member, or the reviews for a location. Most services are essentially large, intelligent caches in front of the database, for example, local LRU, memcached, database calls, in memory partial trees/graphs. Jgroups is occasionally used to keep caches in sync when needed. Most services run 4 instances on different shared/physical machines, sometimes service instances are given their own machines.
- The set of content returned from the service calls is massaged and is organized by the Java code into sets of objects that are passed to a Velocity template (kind of like a weak JSP)
- There is a variety of logic to select different forms of the servlet and/or the Velocity templates based on context which could include POS, language, features, etc. There is also infrastructure to select different code and template paths for A/B and other testing
- Our web servers are arranged in "pools" to allow for A/B testing, and control over deployment. Our weekly release process will deploy to a small pool and let the code run for a few hours to make sure everything is working properly before deploying to all pools

Tech

- Redundant pairs of BigIP, Cisco routers
- Bank of 64 stateless web servers, Linux/Apache/Tomcat running Java servlets, processing 200M+ requests a day
- Bank of 40 stateless service machines running ~100 instances of ~25 services, Jetty/Java, processing 700M+ requests a day

- Postgres, running on 6 pairs (DRDB) machines, processing 700M+ queries a day
- Memcached, 3 separate clusters, running on web servers, 350GB+. Configurable pools of Spy memcache client - asynchronous, NIO to scale requests. Optimization and other changes were done to spy java client to achieve scale and reliability.
- Lucene, wrapped in our service infrastructure, 50M documents, 55GB indexes 8M searches a day, daily incremental updates, complete regeneration once a week. - Hide quoted text -
- JGroups, when there is no other choice, for state synchronization between service machines.
- Hadoop, 16 node cluster with 192TB raw disk storage, 192CPU, 10GB network
- Java, Python, Ruby, PHP, Perl, etc used for tools and supporting infrastructure
- Monitoring - cacti, nagios, custom.
- Two data centers, different cities, in N+1 configuration, one taking traffic in R/W mode, the other in sync and in R/O mode, ready for fail-over 24/7, regular quarterly swap of active
- Total of about 125 machines in each datacenter, all standard Dell equipement
- Logging - 120GB (compressed) application logs, apache access logs, redundant logging of financially sensitive data - both streaming (scribe) and non streaming

Development

- Fully loaded Mac or Linux machines, SVN, Bugzilla, 30" monitors
- Hundreds of virtualized dev servers. Dedicated one per engineer, plus extras on demand
- Weekly deployment cycle - entire code base gets released every Monday
- Daily patches for those "need to get out today" features/fixes
- Over 50 concurrent projects being worked on at a time by ~100 engineers, ~25 get deployed each week
- Engineers work end to end - Design, Code, Test, Monitor. You design something, you code it. You code something, you test it.
- Engineers work across entire stack - HTML, CSS, JS, Java, scripting. If you do not know something, you learn it.
- Release process is shared and rotated among various senior engineers and engineering managers
- Numerous test frameworks available: Unit, Functional, Load, Smoke, Selenium, Load, and a test lab. Its your code, choose what works best for you.
- Numerous mechanisms to help you deliver top quality features: Design Review, Code Review, Deployment Review, Operational Review

Culture

- TripAdvisor engineering can be best compared to running two dozen simultaneous startups, all working on the same code base and running in a common distributed computing environment. Each of these teams has their own business objectives, and each team is able to, and responsible for, all aspects of their business. The only thing that gets in the way of delivering your project is you, as you are expected to work at all levels - design, code, test, monitoring, CSS, JS, Java, SQL, scripting.
- TripAdvisor engineering is organized by business function - there are over two dozen "teams", each responsible for working directly with their business counterparts on SEM, SEO, Mobile, Commerce, CRM, Content, Social Applications, Community, Membership, Sales and Marketing Solutions, China, APAC, Business Listings, Vacation Rentals, Flights, Content Syndication, and others. We do not have the traditional Architect, Coder, QA roles
- Our Operations team is one team that is responsible for the platform that all of these other teams use: Datacenter, Software infrastructure, DevOps, Warehousing. You can think of Operations as our internal "AWS", delivering our 24/7 distributed compute infrastructure as a service, with code/dev/test/deploy all rolled into one. This team includes two technical operations engineers and two site operatons engineers who are responsible for the datacenters and software infrastructure.
- Each of the teams operates in the way that best fits their distinct business and personal needs, our process is best described as "post agile/scrum".
- Culture of responsibility - you own your project end to end, and are responsible for design, coding, testing, monitoring. Most projects are 1-2 engineers.
- Log and measure - tons of data, lots of metrics
- Hackers Week - every engineer gets one week per year to work on any project they want. You can team up with others to tackle larger projects
- Engineering swaps. Pairs of engineers from different teams will swap positions for a few weeks to distribute knowledge and culture
- Web Engineering Program. A new program for engineers who want to work a few months in many different teams.
- Summer Fridays - time shift your weeks during the summer and free up your Fridays
- Yearly charity day - the entire company goes out and contributes their day to a local charity - painting, gardening, etc
- TripAdvisor Charitable Foundation. Funded with several million dollars, employees can apply for grants to a charitable organization that they are personally involved in.

Random thoughts on what we have learned, how we work

- Scalability starts with your culture - how you hire, who you hire, and the expectations that you set.

- Engineers. Hire smart, fast, flexible engineers who are willing to do any type of work, and are excited to learn new technologies. We do not have "architects" - at TripAdvisor, if you design something, your code it, and if you code it you test it. Engineers who do not like to go outside their comfort zone, or who feel certain work is "beneath" them will simply get in the way.
- Hire people who get enjoyment out of delivering things that work. Technology is a means to an end, not an end to itself.
- You own your code and its effects - you design, you test, you code, you monitor. If you break something, you fix it.
- It is better to deliver 20 projects with 10 bugs and miss 5 projects by two days than to deliver 10 projects that are all perfect and on time.
- Encourage learning and pushing the envelope - everyone who works here will make a number of mistakes the first few months, and will continue to occasionally do so over time. The important thing is how much you have learned and to not make the same mistakes over and over again.
- Keep designs simple and focused on the near term business needs - do not design too far ahead. For example, we have rewritten our members functionality as we scaled from tens of thousands, to millions, to tens of millions. We would have done a poor job of designing for tens of millions when all we needed was tens of thousands. We delivered sooner with less problems at each stage, with the cost of the rewrites is small in comparison to what we learned at each stage.
- You can go very far in scaling a database vertically, especially if you minimize use of joins, and especially if you can fit everything into RAM.
- Shard only when necessary, and keep it simple. Our largest single table has well over 1B rows, and it easily scaled vertically until we needed to update/insert tens of millions of rows a day, hitting write IOP limits. At that point, we sharded it at the service level by splitting it into 12 tables, and currently run it on 2 machines each with 6 tables. We can easily scale it to 3, 4 ,6, and then 12 machines without changing our hash algorithm or data distribution, simply by copying tables. We have not had any measurable performance degradation (read or write), and the code to do this was small, easy to understand, and easy to debug. With over 700M database operations a day, we are not anywhere close to sharding any other tables.
- Avoid joins when possible. Our content types (member, media, reviews, etc) are in separate databases, sometimes on shared machines, sometimes on its own machine. It is far better to do two queries (get the set of reviews with their member ids, then get all of the member from this set of ids and merge it at the app level) than do a join. By having the data in different databases, it is easy to scale to one machine per database. It is also easier to keep your content type scalable - we can add new content types in a very modular manner as each content type stands alone. This also aligns well with our service oriented approach, where a service is supported by a database.
- Put end to end responsibility on a single engineer. When one person owns everything (CSS, JS, Java, SQL, scripting), there is no waiting, bottlenecks, scheduling conflicts,

management overhead, or distribution of "ownership". More projects/people can be added in a modular way without affecting everyone else.

- Services. Having a known set of chunky (optimized for the wire) protocols that are aligned to the business and usage patterns makes assembling pages easier, and allows you to scale out each service according to business needs. Big increase in search traffic ? Add more search servers. Also makes you think more carefully about usage patterns and your business.
- Hardware - keep it really really simple. No fancy hardware - no SAN, proprietary devices (aside for networking equipment) - we run all commodity Dell. Assume any components will fail at any time and either have N+1 design or ample resources to make up for failures. Our bank of web servers can handle significant numbers failing - up to 50%. Databases are N+1 with duplicate hot (DRDB based) failover. Services run multiple instances on multiple machines. Load balancer and router each have a hot spare and auto-failover. We have two entire datacenters in different cities, one in active R/W mode handling all the traffic, the other in up to date, R/O mode ready for traffic at any time. We "fail over" every three months to insure the "backup" is ready at all times, and to provide for our continuous/incremental datacenter maintenance.
- Software - keep it really, really, really simple. There are systems you do not want to write yourself - Apache, Tomcat, Jetty, Lucene, Postgres, memcached, Velocity. We have built everything else ourselves - distributed service architecture, web framework, etc. It is not hard to do, and you understand and control everything.
- Process. Less is better. You need to use source code control. You need to be a good code citizen. You need to deliver code that works. You need to communicate your designs and their level of effort. Not much else is "needed" or "required". If you are smart, you will get your design reviewed, your code reviewed, and you will write tests and appropriate monitoring scripts. Hire people who understand that you want these things because they help you deliver better products, not because they are "required". If you make a mistake, and you will, own up to it, and get it fixed. It is also important to find your own mistakes, not rely on others to find them for you.
- Design Reviews. All engineers are invited to a weekly design review. If you have a project that is going to impact others (database, memory usage, new servlets, new libraries, anything of significance) you are expected to present your design at design review and discuss it. This is not only a great way to provide guidance and oversight over the entire system, it is a great way for everyone to learn from each other and be aware of what is going on.