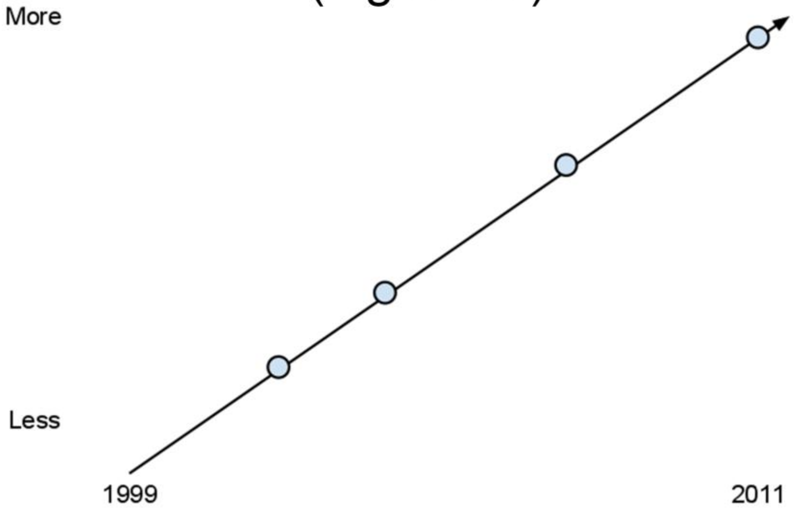
Go Fast and Don't Break Things: Ensuring Quality in the Cloud

Scott Hansma
CTO, Core
Salesforce.com
shansma@salesforce.com
HPTS 2011

Safe Harbor

The requisite growth slide (log scale)

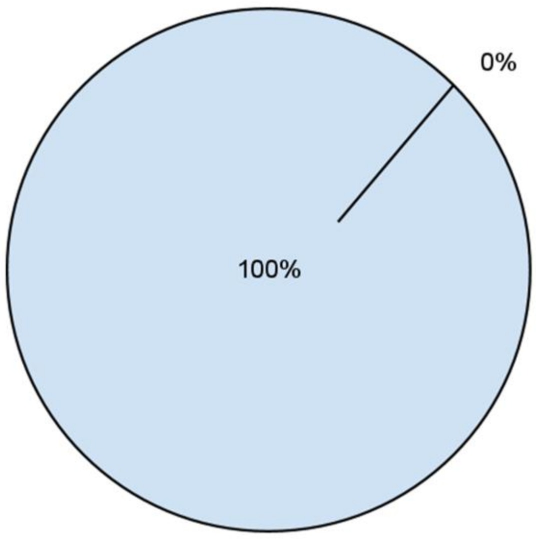


Every industry presentation has to start with a slide showing the staggering scale they've grown to, so here it is.

Background on SFDC:
• Started as a CRM app for sales reps. Always hosted, always multi-tenant.
• Hosted & multi-tenant forced us to provide good APIs and good ways of customizing (e.g. custom schema through flex fields)
• Above got good enough that we're expanded it into a pretty successful platform business, as well as expanding into other areas like customer service

dots:
- when i started at the company / when we pissed off the dalai lama and arnold schwarzenegger in the same quarter as part of our 100k users celebrations.
- first time i met someone who used sfdc
- when i started just assuming everyone used it
- ceo of much larger company spent 30 minutes of his own keynote bashing us / 575 M transactions per day.

Why Quality is More Important in the Cloud



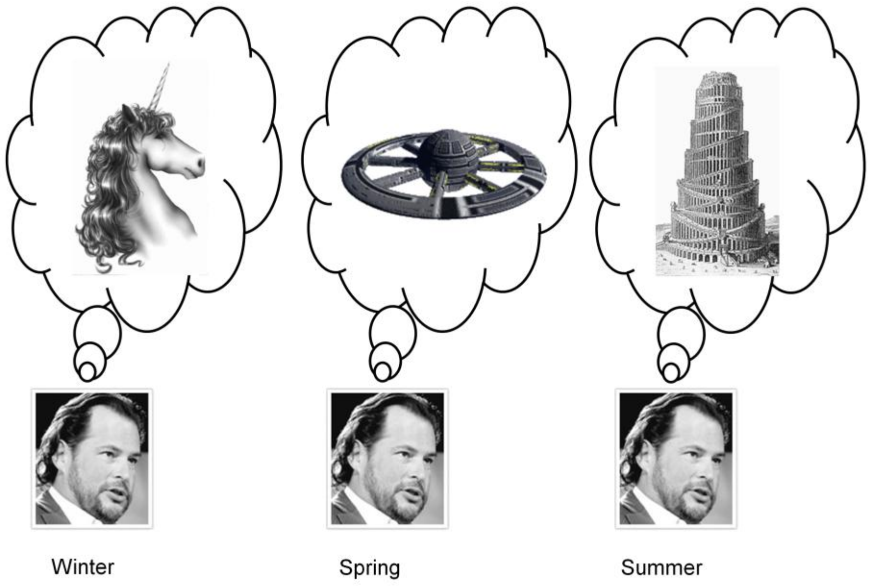
• 100% of customers on shared instances means good things:

- 0% dev time spent maintaining forks
- 100% of features developed for one customer features available for 100% of customers (if they pay for it)

But:
• 0% customers get to opt out of a new version (so we'd better get it right!)

Also:
- 100% (-ish) of our money comes from customers paying us (i.e., not ads etc) and they will pay us 0(%) if they don't like us. (So we'd better get it right!)

And we deliver 3 major releases per year



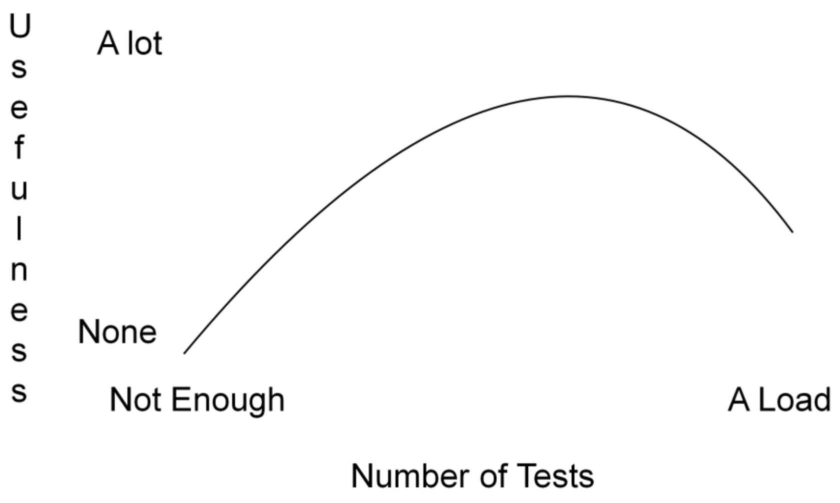
The web 2.0 kids may not think of 3 major releases per year as a lot, but for enterprise software, it’s unheard-of. And even for web 2.0, the amount of change in each release is pretty Impressive.

For example Chatter – our CEO (pictured) decided we should build an employee social network into our CRM product, so we did.

And we do 400 minor releases a year.

Cool Things that Everyone Can (Should?) Do

Too many tests is a problem too.
And it's a hard problem.



We started out with no tests (don’t judge – it was 1999).

Then we found religion and wrote a bunch of tests. Regressions plummet, life is great.

Then we pass some tipping point where we have so many tests (including some poorly written ones) that every test run produces a ton of failures. Devs stop paying attention. Regressions skyrocket, life is bad.

And to make matters worse, we have so many tests that we can’t keep adding hardware fast enough to run them all.

So we developed a system that eventually grew into…

- Run entire suite of tests on *n* changelists
  - Multiple suites along spectrum of fast-complete.
  - Parallelized across 10-80 VMs (in DB + app pairs) depending on suite size.
- Compare results to previous run.
- For the set of newly failing tests:
  - Rerun immediately. If a test now passes, it's a flapper. File a bug to test author.
  - Binary search across sets of changelists to pinpoint cause of each test failure. File a bug to changelist author (or re-open closed bug).
- For all newly passing tests:
  - Mark the associated bug fixed (except flappers)
- Throw away all the VMs
  - So tests always start in a clean state.

Get excited! A slide with actual content!

It should have been a flowchart, but I hate drawing flowcharts in PPT.

(walk through bullets)

This greatly reduces the capacity problem by reducing the amount of work needed to run all tests. Doesn't even trade off time to results because you can now parallelize the test running.

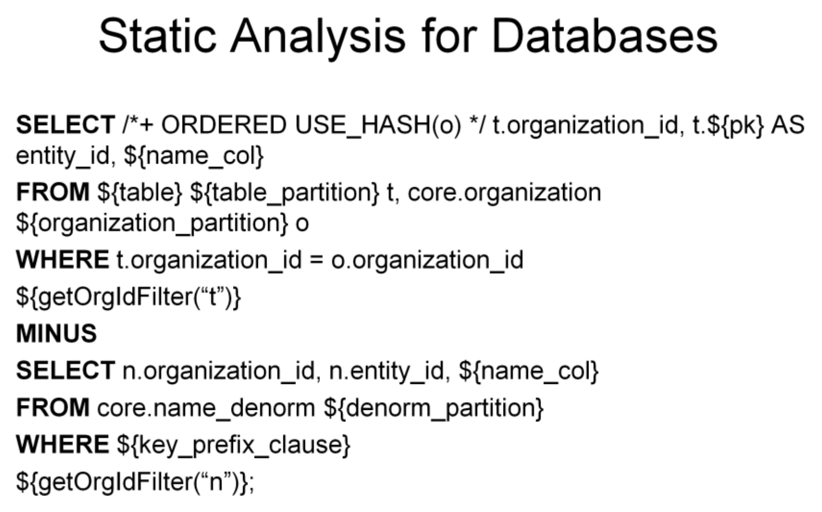
"Flapper" = poorly written test that "flaps" between passing and failing. Some common causes are bad concurrency design in test, timeouts, assumptions about contents of database.

This solves the relevancy problem by only showing devs the tests they broke.

(done walking through bullets)

Numbers: Approx 150k test cases, 600 checkins per day, 3000 VMs. Currently on VMWare because we started this whole thing before AWS, but we have an AWS prototype running.

All custom built because we started work on it before Hudson/Jenkins was around, but also because as far as we can tell, this system is pretty novel with the tight bug integration and the failure pinpointing and all that. If we were starting today, we'd probably base it on Jenkins, but right now there's not a sufficient ROI for converting.



Static Analysis for Databases

```
SELECT /*+ ORDERED USE_HASH(o) */ t.organization_id, t.${pk} AS
entity_id, ${name_col}
FROM ${table} ${table_partition} t, core.organization
${organization_partition} o
WHERE t.organization_id = o.organization_id
${getOrgIdFilter("t")}
MINUS
SELECT n.organization_id, n.entity_id, ${name_col}
FROM core.name_denorm ${denorm_partition}
WHERE ${key_prefix_clause}
${getOrgIdFilter("n")};
```

Tests are great but you typically have to write new tests for each new piece of code.

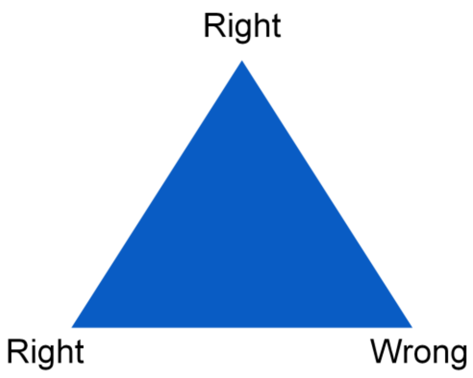
Which is why we all know static analysis is awesome. We think it is too. We use FindBugs for Java just like everyone else and it’s great.

Scrutiny: our own tool. Database invariants in the form of queries that should return 0 rows. This example here is for checking that our denormalized table for name lookup is in sync with all the main tables. It’s parameterized such that you can run it for a particular organization (our term for tenant) or a particular database partition or the whole database. We run these in production but also after every functional test class to find bugs the developers didn’t think to test for. Finds a ton of stuff. Downside is that devs sometimes don’t take the failures seriously, so it really benefits from the targeted assignment described above.

We’ve also written a FindBugs equivalent for our stored procedure code (which we have a ton of). It checks for stuff like properly hinting and using indexes, as well as salesforce-specific things like keeping pairs of related columns in sync and properly using session state. It also runs on dynamic SQL at runtime during functional test runs, which is cool.

Every good presentation needs one of those "pick 2 of 3" engineering trade offs, which I have in the next slide.

API Versions and Gold Files



Pick any
Two

This next one is a little more subtle. Any behavior change in a public API, **even to fix a bug** can break a customer’s integration, and thus is a bug in itself.

To reiterate from the start of the talk: this is critically important for cloud providers because customers can’t opt out of a release, so we can’t ask them to change their integration before they upgrade.

Every API is versioned. No destructive behavior changes without a version number change. We support old APIs for far far too long.

Gold files: Query all our APIs and all versions and dump the results to a file.

We run that test on every checkin and make a human approve any changes by checking in a new gold file. (Valid changes include new fields or objects added to APIs, potentially other changes depending on the API’s contract).

Cool Things that Only Cloud Providers Can Do

Before the Release: Apex & Report Hammers

Image: scrapstothefuture/flickr

Apex is our stored procedure language. If I say call it a “stored procedure language” then technical types don’t ask the completely reasonable “why the hell would you write your own language” but marketing types freak out because you could write your whole app in Apex (and people do).

Something awesome you can only do as a cloud vendor: Run ALL customer-written apex tests with and without major code change. For this release we’re piloting a complete rewrite of our interpreter and it found over 500 issues that we fixed before production. Last release was over 300 bugs.

Same story for our report hammer, which runs all the reports that all our customers have defined to look for performance (often an issue) and correctness (rarely an issue).

Runs on copy of production data (in prod data centers; no dev access to customer data!). Devs only see problem/no problem, not results of reports or Apex tests. Doesn’t run with every checkin. That would be nice but basically impossible (we’d have to upgrade 100s of TB of relational databases with every checkin!!)

Reiterate: as a customer, all your tests get run before an upgrade and then we fix any bugs that we uncover before we release. That’s pretty damn cool, dontcha think? Don’t you wish your OS/Database/etc vendor did that for you?

After the Release: Gack Watching

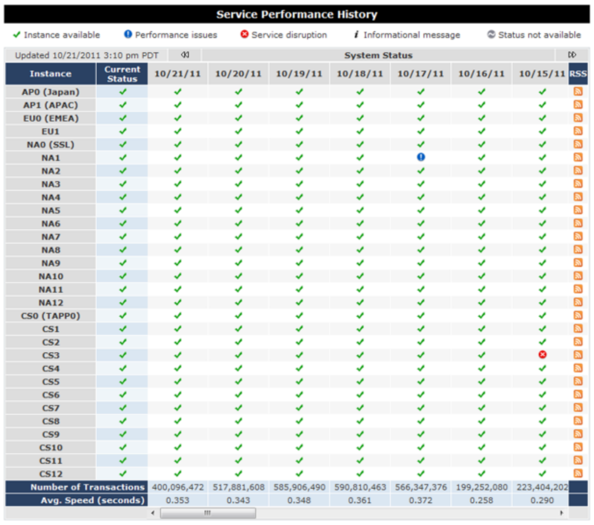
Image: Dylan Ashe

"Gack" = salesforce term for exception in production; from our founder's mis-remembering of what Bill the Cat says (he actually says "Ack").

Whenever a new exception happens in production, we automatically file a bug and so can fix it without customers seeing the issue.

We've spent a bunch of time tweaking the exact mapping from exception to gack. Most exceptions are a real problem and we need to fix it. Some (for example connection resets while we're taking down a server) we suppress altogether. Others (socket timeouts that will be retried) we log so that if there's a problem we can see the details, but either don't alert on because it's not a problem or only alert above a threshhold. Lots of people have written about the ins and outs of setting up production monitoring and alerts.

Keeping us Committed: trust.salesforce.com



Service Performance History

✓ Instance available ● Performance issues ⊗ Service disruption i Informational message ⟳ Status not available

Updated 10/21/2011 3:10 pm PDT — System Status

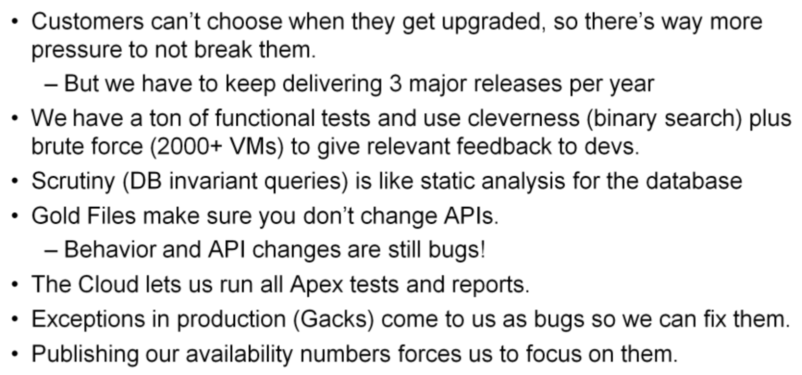
Instance	Current Status	10/21/11	10/20/11	10/19/11	10/18/11	10/17/11	10/16/11	10/15/11	RSS
AP0 (Japan)	✓	✓	✓	✓	✓	✓	✓	✓	
AP1 (APAC)	✓	✓	✓	✓	✓	✓	✓	✓	
EU0 (EMEA)	✓	✓	✓	✓	✓	✓	✓	✓	
EU1	✓	✓	✓	✓	✓	✓	✓	✓	
NA0 (SSL)	✓	✓	✓	✓	✓	✓	✓	✓	
NA1	✓	✓	✓	✓	✓	●	✓	✓	
NA2	✓	✓	✓	✓	✓	✓	✓	✓	
NA3	✓	✓	✓	✓	✓	✓	✓	✓	
NA4	✓	✓	✓	✓	✓	✓	✓	✓	
NA5	✓	✓	✓	✓	✓	✓	✓	✓	
NA6	✓	✓	✓	✓	✓	✓	✓	✓	
NA7	✓	✓	✓	✓	✓	✓	✓	✓	
NA8	✓	✓	✓	✓	✓	✓	✓	✓	
NA9	✓	✓	✓	✓	✓	✓	✓	✓	
NA10	✓	✓	✓	✓	✓	✓	✓	✓	
NA11	✓	✓	✓	✓	✓	✓	✓	✓	
NA12	✓	✓	✓	✓	✓	✓	✓	✓	
CS0 (TAPP0)	✓	✓	✓	✓	✓	✓	✓	✓	
CS1	✓	✓	✓	✓	✓	✓	✓	✓	
CS2	✓	✓	✓	✓	✓	✓	✓	✓	
CS3	✓	✓	✓	✓	✓	✓	✓	⊗	
CS4	✓	✓	✓	✓	✓	✓	✓	✓	
CS5	✓	✓	✓	✓	✓	✓	✓	✓	
CS6	✓	✓	✓	✓	✓	✓	✓	✓	
CS7	✓	✓	✓	✓	✓	✓	✓	✓	
CS8	✓	✓	✓	✓	✓	✓	✓	✓	
CS9	✓	✓	✓	✓	✓	✓	✓	✓	
CS10	✓	✓	✓	✓	✓	✓	✓	✓	
CS11	✓	✓	✓	✓	✓	✓	✓	✓	
CS12	✓	✓	✓	✓	✓	✓	✓	✓	
Number of Transactions		400,096,472	517,881,608	585,906,490	590,810,463	566,347,376	199,252,080	223,404,202	
Avg. Speed (seconds)		0.353	0.343	0.348	0.361	0.372	0.258	0.290	

- Created in response to serious site problems back in 2006.
- Made customers feel good because they could see how we were doing against our promise to work on problems.
- More important: forces us to pay attention to availability and performance ongoing; otherwise it's easy for the company to let it slide

- Nice side benefit: makes outages not news

- You'll notice there's one blue dot and one red dot in the picture. Red dot means some kind of service disruption. Blue dot means performance issue, which we take as seriously as outages. We feel bad about these dots, but we're open about it because it builds trust in the cloud. And because of the way our service is architected, each of those dots only corresponds to maybe 5% of our customers, so that's nice too. But that's another talk.
]

Thank you



- Customers can't choose when they get upgraded, so there's way more pressure to not break them.
  - – But we have to keep delivering 3 major releases per year
- We have a ton of functional tests and use cleverness (binary search) plus brute force (2000+ VMs) to give relevant feedback to devs.
- Scrutiny (DB invariant queries) is like static analysis for the database
- Gold Files make sure you don't change APIs.
  - – Behavior and API changes are still bugs!
- The Cloud lets us run all Apex tests and reports.
- Exceptions in production (Gacks) come to us as bugs so we can fix them.
- Publishing our availability numbers forces us to focus on them.

Summary slide

Any questions?