On Designing and Deploying Internet-Scale Services

James Hamilton – Windows Live Services Platform

ABSTRACT

The system-to-administrator ratio is commonly used as a rough metric to understand administrative costs in high-scale services. With smaller, less automated services this ratio can be as low as 2:1, whereas on industry leading, highly automated services, we've seen ratios as high as 2,500:1. Within Microsoft services, Autopilot [1] is often cited as the magic behind the success of the Windows Live Search team in achieving high system-to-administrator ratios. While auto-administration is important, the most important factor is actually the service itself. Is the service efficient to automate? Is it what we refer to more generally as operations-friendly? Services that are operations-friendly require little human intervention, and both detect and recover from all but the most obscure failures without administrative intervention. This paper summarizes the best practices accumulated over many years in scaling some of the largest services at MSN and Windows Live.

Introduction

This paper summarizes a set of best practices for designing and developing operations-friendly services. Designing and deploying high-scale services is a rapidly evolving subject area and, consequently, any list of best practices will likely grow and morph over time. Our aim is to help others

1. deliver operations-friendly services quickly and
2. avoid the early morning phone calls and meetings with unhappy customers that non-operations-friendly services tend to yield.

The work draws on our experiences over the last 20 years in high-scale data-centric software systems and internet-scale services, most recently from leading the Exchange Hosted Services team (at the time, a mid-sized service of roughly 700 servers and just over 2.2M users). We also incorporate the experiences of the Windows Live Search, Windows Live Mail, Exchange Hosted Services, Live Communications Server, Windows Live Address Book Clearing House (ABCH), MSN Spaces, Xbox Live, Rackable Systems Engineering Team, and the Messenger Operations teams in addition to that of the overall Microsoft Global Foundation Services Operations team. Several of these contributing services have grown to more than a quarter billion users. The paper also draws heavily on the work done at Berkeley on Recovery Oriented Computing [2, 3] and at Stanford on Crash-Only Software [4, 5].

Bill Hoffman [6] contributed many best practices to this paper, but also a set of three simple tenets worth considering up front:

1. Expect failures. A component may crash or be stopped at any time. Dependent components might fail or be stopped at any time. There will be network failures. Disks will run out of space. Handle all failures gracefully.
2. Keep things simple. Complexity breeds problems. Simple things are easier to get right. Avoid unnecessary dependencies. Installation should be simple. Failures on one server should have no impact on the rest of the data center.
3. Automate everything. People make mistakes. People need sleep. People forget things. Automated processes are testable, fixable, and therefore ultimately much more reliable. Automate wherever possible.

These three tenets form a common thread throughout much of the discussion that follows.

Recommendations

This section is organized into ten sub-sections, each covering a different aspect of what is required to design and deploy an operations-friendly service. These sub-sections include overall service design; designing for automation and provisioning; dependency management; release cycle and testing; hardware selection and standardization; operations and capacity planning; auditing, monitoring and alerting; graceful degradation and admission control; customer and press communications plan; and customer self provisioning and self help.

Overall Application Design

We have long believed that 80% of operations issues originate in design and development, so this section on overall service design is the largest and most important. When systems fail, there is a natural tendency to look first to operations since that is where the problem actually took place. Most operations issues, however, either have their genesis in design and development or are best solved there.

Throughout the sections that follow, a consensus emerges that firm separation of development, test, and operations isn't the most effective approach in the services world. The trend we've seen when looking across many services is that low-cost administration correlates highly with how closely the development, test, and operations teams work together.

In addition to the best practices on service design discussed here, the subsequent section, "Designing for Automation Management and Provisioning," also has substantial influence on service design. Effective automatic management and provisioning are generally achieved only with a constrained service model. This is a repeating theme throughout: simplicity is the key to efficient operations. Rational constraints on hardware selection, service design, and deployment models are a big driver of reduced administrative costs and greater service reliability.

Some of the operations-friendly basics that have the biggest impact on overall service design are:

- **Design for failure**. This is a core concept when developing large services that comprise many cooperating components. Those components will fail and they will fail frequently. The components don't always cooperate and fail independently either. Once the service has scaled beyond 10,000 servers and 50,000 disks, failures will occur multiple times a day. If a hardware failure requires any immediate administrative action, the service simply won't scale cost-effectively and reliably. The entire service must be capable of surviving failure without human administrative interaction. Failure recovery must be a very simple path and that path must be tested frequently. Armando Fox of Stanford [4, 5] has argued that the best way to test the failure path is never to shut the service down normally. Just hard-fail it. This sounds counter-intuitive, but if the failure paths aren't frequently used, they won't work when needed [7].
- **Redundancy and fault recovery**. The mainframe model was to buy one very large, very expensive server. Mainframes have redundant power supplies, hot-swappable CPUs, and exotic bus architectures that provide respectable I/O throughput in a single, tightly-coupled system. The obvious problem with these systems is their expense. And, even with all the costly engineering, they still aren't sufficiently reliable. In order to get the fifth 9 of reliability, redundancy is required. Even getting four 9's on a single-system deployment is difficult. This concept is fairly well understood industry-wide, yet it's still common to see services built upon fragile, non-redundant data tiers.

  Designing a service such that any system can crash (or be brought down for service) at any time while still meeting the service level agreement (SLA) requires careful engineering. The acid test for full compliance with this design principle is the following: is the operations team willing and able to bring down any server in the service at any time without draining the work load first? If they are, then there is synchronous redundancy (no data loss), failure detection, and automatic take-over. As a design approach, we recommend one commonly used to find and correct potential service security issues: security threat modeling. In security threat modeling, we consider each possible security threat and, for each, implement adequate mitigation. The same approach can be applied to designing for fault resiliency and recovery.

  Document all conceivable component failures modes and combinations thereof. For each failure, ensure that the service can continue to operate without unacceptable loss in service quality, or determine that this failure risk is acceptable for this particular service (e.g., loss of an entire data center in a non-geo-redundant service). Very unusual combinations of failures may be determined sufficiently unlikely that ensuring the system can operate through them is uneconomical. Be cautious when making this judgment. We've been surprised at how frequently "unusual" combinations of events take place when running thousands of servers that produce millions of opportunities for component failures each day. Rare combinations can become commonplace.
- **Commodity hardware slice**. All components of the service should target a commodity hardware slice. For example, storage-light servers will be dual socket, 2- to 4-core systems in the $1,000 to $2,500 range with a boot disk. Storage-heavy servers are similar servers with 16 to 24 disks. The key observations are
  1. large clusters of commodity servers are much less expensive than the small number of large servers they replace,
  2. server performance continues to increase much faster than I/O performance, making a small server a more balanced system for a given amount of disk,
  3. power consumption scales linearly with servers but cubically with clock frequency, making higher performance servers more expensive to operate, and
  4. a small server affects a smaller proportion of the overall service workload when failing over.
- **Single-version software**. Two factors that make some services less expensive to develop and faster to evolve than most packaged products are
  - the software needs to only target a single internal deployment and
  - previous versions don't have to be supported for a decade as is the case for enterprise-targeted products.

  Single-version software is relatively easy to achieve with a consumer service, especially one provided without charge. But it's equally important when selling subscription-based services to

non-consumers. Enterprises are used to having significant influence over their software providers and to having complete control over when they deploy new versions (typically slowly). This drives up the cost of their operations and the cost of supporting them since so many versions of the software need to be supported.

The most economic services don't give customers control over the version they run, and only host one version. Holding this single-version software line requires

1. care in not producing substantial user experience changes release-to-release and
2. a willingness to allow customers that need this level of control to either host internally or switch to an application service provider willing to provide this people-intensive multi-version support.

- **Multi-tenancy**. Multi-tenancy is the hosting of all companies or end users of a service in the same service without physical isolation, whereas single tenancy is the segregation of groups of users in an isolated cluster. The argument for multi-tenancy is nearly identical to the argument for single version support and is based upon providing fundamentally lower cost of service built upon automation and large-scale.

In review, the basic design tenets and considerations we have laid out above are:

- design for failure,
- implement redundancy and fault recovery,
- depend upon a commodity hardware slice,
- support single-version software, and
- implement multi-tenancy.

We are constraining the service design and operations model to maximize our ability to automate and to reduce the overall costs of the service. We draw a clear distinction between these goals and those of application service providers or IT outsourcers. Those businesses tend to be more people intensive and more willing to run complex, customer specific configurations.

More specific best practices for designing operations-friendly services are:

- **Quick service health check**. This is the services version of a build verification test. It's a sniff test that can be run quickly on a developer's system to ensure that the service isn't broken in any substantive way. Not all edge cases are tested, but if the quick health check passes, the code can be checked in.
- **Develop in the full environment**. Developers should be unit testing their components, but should also be testing the full service with their component changes. Achieving this goal efficiently requires single-server deployment (section 2.4), and the preceding best practice, a quick service health check.
- **Zero trust of underlying components**. Assume that underlying components will fail and ensure that components will be able to recover and continue to provide service. The recovery technique is service-specific, but common techniques are to
  - continue to operate on cached data in read-only mode or
  - continue to provide service to all but a tiny fraction of the user base during the short time while the service is accessing the redundant copy of the failed component.
- **Do not build the same functionality in multiple components**. Foreseeing future interactions is hard, and fixes have to be made in multiple parts of the system if code redundancy creeps in. Services grow and evolve quickly. Without care, the code base can deteriorate rapidly.
- **One pod or cluster should not affect another pod or cluster**. Most services are formed of pods or sub-clusters of systems that work together to provide the service, where each pod is able to operate relatively independently. Each pod should be as close to 100% independent and without inter-pod correlated failures. Global services even with redundancy are a central point of failure. Sometimes they cannot be avoided but try to have everything that a cluster needs inside the clusters.
- **Allow (rare) emergency human intervention**. The common scenario for this is the movement of user data due to a catastrophic event or other emergency. Design the system to never need human interaction, but understand that rare events will occur where combined failures or unanticipated failures require human interaction. These events will happen and operator error under these circumstances is a common source of catastrophic data loss. An operations engineer working under pressure at 2 a.m. will make mistakes. Design the system to first not require operations intervention under most circumstances, but work with operations to come up with recovery plans if they need to intervene. Rather than documenting these as multi-step, error-prone procedures, write them as scripts and test them in production to ensure they work. What isn't tested in production won't work, so periodically the operations team should conduct a "fire drill" using these tools. If the service-availability risk of a drill is excessively high, then insufficient investment has been made in the design, development, and testing of the tools.
- **Keep things simple and robust**. Complicated algorithms and component interactions multiply the difficulty of debugging, deploying, etc. Simple and nearly stupid is almost always better in a high-scale service-the number of interacting failure modes is already daunting before complex optimizations are delivered. Our general rule is that optimizations that bring an order of magnitude improvement are worth considering, but

percentage or even small factor gains aren't worth it.

- **Enforce admission control at all levels**. Any good system is designed with admission control at the front door. This follows the long-understood principle that it's better to not let more work into an overloaded system than to continue accepting work and beginning to thrash. Some form of throttling or admission control is common at the entry to the service, but there should also be admission control at all major components boundaries. Work load characteristic changes will eventually lead to sub-component overload even though the overall service is operating within acceptable load levels. See the note below in section 2.8 on the "big red switch" as one way of gracefully degrading under excess load. The general rule is to attempt to gracefully degrade rather than hard failing and to block entry to the service before giving uniform poor service to all users.
- **Partition the service**. Partitions should be infinitely-adjustable and fine-grained, and not be bounded by any real world entity (person, collection ...). If the partition is by company, then a big company will exceed the size of a single partition. If the partition is by name prefix, then eventually all the P's, for example, won't fit on a single server. We recommend using a look-up table at the mid-tier that maps fine-grained entities, typically users, to the system where their data is managed. Those fine-grained partitions can then be moved freely between servers.
- **Understand the network design**. Test early to understand what load is driven between servers in a rack, across racks, and across data centers. Application developers must understand the network design and it must be reviewed early with networking specialists on the operations team.
- **Analyze throughput and latency**. Analysis of the throughput and latency of core service user interactions should be performed to understand impact. Do so with other operations running such as regular database maintenance, operations configuration (new users added, users migrated), service debugging, etc. This will help catch issues driven by periodic management tasks. For each service, a metric should emerge for capacity planning such as user requests per second per system, concurrent on-line users per system, or some related metric that maps relevant work load to resource requirements.
- **Treat operations utilities as part of the service**. Operations utilities produced by development, test, program management, and operations should be code-reviewed by development, checked into the main source tree, and tracked on the same schedule and with the same testing. Frequently these utilities are mission critical and yet nearly untested.
- **Understand access patterns**. When planning new features, always consider what load they are going to put on the backend store. Often the service model and service developers become so abstracted away from the store that they lose sight of the load they are putting on the underlying database. A best practice is to build it into the spec with a section such as. "What impacts will this feature have on the rest of the infrastructure?" Then measure and validate the feature for load when it goes live.
- **Version everything**. Expect to run in a mixed-version environment. The goal is to run single version software but multiple versions will be live during rollout and production testing. Versions n and n+1 of all components need to co-exist peacefully.
- **Keep the unit/functional tests from the last release**. These tests are a great way of verifying that version n-1 functionality doesn't get broken. We recommend going one step further and constantly running service verification tests in production (more detail below).
- **Avoid single points of failure**. Single points of failure will bring down the service or portions of the service when they fail. Prefer stateless implementations. Don't affinitize requests or clients to specific servers. Instead, load balance over a group of servers able to handle the load. Static hashing or any static work allocation to servers will suffer from data and/or query skew problems over time. Scaling out is easy when machines in a class are interchangeable. Databases are often single points of failure and database scaling remains one of the hardest problems in designing internet-scale services. Good designs use fine-grained partitioning and don't support cross-partition operations to allow efficient scaling across many database servers. All database state is stored redundantly (on at least one) fully redundant hot standby server and failover is tested frequently in production.

Automatic Management and Provisioning

Many services are written to alert operations on failure and to depend upon human intervention for recovery. The problem with this model starts with the expense of a 24x7 operations staff. Even more important is that if operations engineers are asked to make tough decisions under pressure, about 20% of the time they will make mistakes. The model is both expensive and error-prone, and reduces overall service reliability.

Designing for automation, however, involves significant service-model constraints. For example, some of the large services today depend upon database systems with asynchronous replication to a secondary, back-up server. Failing over to the secondary after the

primary isn't able to service requests loses some customer data due to replicating asynchronously. However, not failing over to the secondary leads to service downtime for those users whose data is stored on the failed database server. Automating the decision to fail over is hard in this case since its dependent upon human judgment and accurately estimating the amount of data loss compared to the likely length of the down time. A system designed for automation pays the latency and throughput cost of synchronous replication. And, having done that, failover becomes a simple decision: if the primary is down, route requests to the secondary. This approach is much more amenable to automation and is considerably less error prone.

Automating administration of a service after design and deployment can be very difficult. Successful automation requires simplicity and clear, easy-to-make operational decisions. This in turn depends on a careful service design that, when necessary, sacrifices some latency and throughput to ease automation. The trade-off is often difficult to make, but the administrative savings can be more than an order of magnitude in high-scale services. In fact, the current spread between the most manual and the most automated service we've looked at is a full two orders of magnitude in people costs.

Best practices in designing for automation include:

- **Be restartable and redundant**. All operations must be restartable and all persistent state stored redundantly.
- **Support geo-distribution**. All high scale services should support running across several hosting data centers. In fairness, automation and most of the efficiencies we describe here are still possible without geo-distribution. But lacking support for multiple data center deployments drives up operations costs dramatically. Without geo-distribution, it's difficult to use free capacity in one data center to relieve load on a service hosted in another data center. Lack of geo-distribution is an operational constraint that drives up costs.
- **Automatic provisioning and installation**. Provisioning and installation, if done by hand, is costly, there are too many failures, and small configuration differences will slowly spread throughout the service making problem determination much more difficult.
- **Configuration and code as a unit**. Ensure that
  - the development team delivers the code and the configuration as a single unit,
  - the unit is deployed by test in exactly the same way that operations will deploy it, and
  - operations deploys them as a unit.

  Services that treat configuration and code as a unit and only change them together are often more reliable.
- If a configuration change must be made in production, ensure that all changes produce an audit log record so it's clear what was changed, when and by whom, and which servers were effected (see section 2.7). Frequently scan all servers to ensure their current state matches the intended state. This helps catch install and configuration failures, detects server misconfigurations early, and finds non-audited server configuration changes.
- **Manage server roles or personalities rather than servers**. Every system role or personality should support deployment on as many or as few servers as needed.
- **Multi-system failures are common**. Expect failures of many hosts at once (power, net switch, and rollout). Unfortunately, services with state will have to be topology-aware. Correlated failures remain a fact of life.
- **Recover at the service level**. Handle failures and correct errors at the service level where the full execution context is available rather than in lower software levels. For example, build redundancy into the service rather than depending upon recovery at the lower software layer.
- **Never rely on local storage for non-recoverable information**. Always replicate all the non-ephemeral service state.
- **Keep deployment simple**. File copy is ideal as it gives the most deployment flexibility. Minimize external dependencies. Avoid complex install scripts. Anything that prevents different components or different versions of the same component from running on the same server should be avoided.
- **Fail services regularly**. Take down data centers, shut down racks, and power off servers. Regular controlled brown-outs will go a long way to exposing service, system, and network weaknesses. Those unwilling to test in production aren't yet confident that the service will continue operating through failures. And, without production testing, recovery won't work when called upon.

Dependency Management

Dependency management in high-scale services often doesn't get the attention the topic deserves. As a general rule, dependence on small components or services doesn't save enough to justify the complexity of managing them. Dependencies do make sense when

1. the components being depended upon are substantial in size or complexity, or
2. the service being depended upon gains its value in being a single, central instance.

Examples of the first class are storage and consensus algorithm implementations. Examples of the second class of are identity and group management systems. The whole value of these systems is that they are a

single, shared instance so multi-instancing to avoid dependency isn't an option.

Assuming that dependencies are justified according to the above rules, some best practices for managing them are:

- **Expect latency**. Calls to external components may take a long time to complete. Don't let delays in one component or service cause delays in completely unrelated areas. Ensure all interactions have appropriate timeouts to avoid tying up resources for protracted periods. Operational idempotency allows the restart of requests after timeout even though those requests may have partially or even fully completed. Ensure all restarts are reported and bound restarts to avoid a repeatedly failing request from consuming ever more system resources.
- **Isolate failures**. The architecture of the site must prevent cascading failures. Always "fail fast." When dependent services fail, mark them as down and stop using them to prevent threads from being tied up waiting on failed components.
- **Use shipping and proven components**. Proven technology is almost always better than operating on the bleeding edge. Stable software is better than an early copy, no matter how valuable the new feature seems. This rule applies to hardware as well. Stable hardware shipping in volume is almost always better than the small performance gains that might be attained from early release hardware.
- **Implement inter-service monitoring and alerting**. If the service is overloading a dependent service, the depending service needs to know and, if it can't back-off automatically, alerts need to be sent. If operations can't resolve the problem quickly, it needs to be easy to contact engineers from both teams quickly. All teams with dependencies should have engineering contacts on the dependent teams.
- **Dependent services require the same design point**. Dependent services and producers of dependent components need to be committed to at least the same SLA as the depending service.
- **Decouple components**. Where possible, ensure that components can continue operation, perhaps in a degraded mode, during failures of other components. For example, rather than reauthenticating on each connect, maintain a session key and refresh it every N hours independent of connection status. On reconnect, just use existing session key. That way the load on the authenticating server is more consistent and login storms are not driven on reconnect after momentary network failure and related events.

Release Cycle and Testing

Testing in production is a reality and needs to be part of the quality assurance approach used by all internet-scale services. Most services have at least one test lab that is as similar to production as (affordably) possible and all good engineering teams use production workloads to drive the test systems realistically. Our experience has been, however, that as good as these test labs are, they are never full fidelity. They always differ in at least subtle ways from production. As these labs approach the production system in fidelity, the cost goes asymptotic and rapidly approaches that of the production system.

We instead recommend taking new service releases through standard unit, functional, and production test lab testing and then going into limited production as the final test phase. Clearly we don't want software going into production that doesn't work or puts data integrity at risk, so this has to be done carefully. The following rules must be followed:

1. the production system has to have sufficient redundancy that, in the event of catastrophic new service failure, state can be quickly be recovered,
2. data corruption or state-related failures have to be extremely unlikely (functional testing must first be passing),
3. errors must be detected and the engineering team (rather than operations) must be monitoring system health of the code in test, and
4. it must be possible to quickly roll back all changes and this roll back must be tested before going into production.

This sounds dangerous. But we have found that using this technique actually improves customer experience around new service releases. Rather than deploying as quickly as possible, we put one system in production for a few days in a single data center. Then we bring one new system into production in each data center. Then we'll move an entire data center into production on the new bits. And finally, if quality and performance goals are being met, we deploy globally. This approach can find problems before the service is at risk and can actually provide a better customer experience through the version transition. Big-bang deployments are very dangerous.

Another potentially counter-intuitive approach we favor is deployment mid-day rather than at night. At night, there is greater risk of mistakes. And, if anomalies crop up when deploying in the middle of the night, there are fewer engineers around to deal with them. The goal is to minimize the number of engineering and operations interactions with the system overall, and especially outside of the normal work day, to both reduce costs and to increase quality.

Some best practices for release cycle and testing include:

- **Ship often**. Intuitively one would think that shipping more frequently is harder and more error prone. We've found, however, that more

frequent releases have less big-bang changes. Consequently, the releases tend to be higher quality and the customer experience is much better. The acid test of a good release is that the user experience may have changed but the number of operational issues around availability and latency should be unchanged during the release cycle. We like shipping on 3-month cycles, but arguments can be made for other schedules. Our gut feel is that the norm will eventually be less than three months, and many services are already shipping on weekly schedules. Cycles longer than three months are dangerous.

- **Use production data to find problems**. Quality assurance in a large-scale system is a data-mining and visualization problem, not a testing problem. Everyone needs to focus on getting the most out of the volumes of data in a production environment. A few strategies are:
  - **Measureable release criteria**. Define specific criteria around the intended user experience, and continuously monitor it. If availability is supposed to be 99%, measure that availability meets the goal. Both alert and diagnose if it goes under.
  - **Tune goals in real time**. Rather than getting bogged down deciding whether the goal should be 99% or 99.9% or any other goal, set an acceptable target and then ratchet it up as the system establishes stability in production.
  - **Always collect the actual numbers**. Collect the actual metrics rather than red and green or other summary reports. Summary reports and graphs are useful but the raw data is needed for diagnosis.
  - **Minimize false positives**. People stop paying attention very quickly when the data is incorrect. It's important to not over-alert or operations staff will learn to ignore them. This is so important that hiding real problems as collateral damage is often acceptable.
  - **Analyze trends**. This can be used for predicting problems. For example, when data movement in the system diverges from the usual rate, it often predicts a bigger problem. Exploit the available data.
  - **Make the system health highly visible**. Require a globally available, real-time display of service health for the entire organization. Have an internal website people can go at any time to understand the current state of the service.
  - **Monitor continuously**. It bears noting that people must be looking at all the data every day. Everyone should do this, but make it the explicit job of a subset of the team to do this.
- **Invest in engineering**. Good engineering minimizes operational requirements and solves problems before they actually become operational issues. Too often, organizations grow operations to deal with scale and never take the time to engineer a scalable, reliable architecture. Services that don't think big to start with will be scrambling to catch up later.
- **Support version roll-back**. Version roll-back is mandatory and must be tested and proven before roll-out. Without roll-back, any form of production-level testing in very high risk. Reverting to the previous version is a rip cord that should always be available on any deployment.
- **Maintain forward and backward compatibility**. This vital point strongly relates to the previous one. Changing file formats, interfaces, logging/debugging, instrumentation, monitoring and contact points between components are all potential risk. Don't rip out support for old file formats until there is no chance of a roll back to that old format in the future.
- **Single-server deployment**. This is both a test and development requirement. The entire service must be easy to host on a single system. Where single-server deployment is impossible for some component (e.g., a dependency on an external, non-single box deployable service), write an emulator to allow single-server testing. Without this, unit testing is difficult and doesn't fully happen. And if running the full system is difficult, developers will have a tendency to take a component view rather than a systems view.
- **Stress test for load**. Run some tiny subset of the production systems at twice (or more) the load to ensure that system behavior at higher than expected load is understood and that the systems don't melt down as the load goes up.
- **Perform capacity and performance testing prior to new releases**. Do this at the service level and also against each component since work load characteristics will change over time. Problems and degradations inside the system need to be caught early.
- **Build and deploy shallowly and iteratively**. Get a skeleton version of the full service up early in the development cycle. This full service may hardly do anything at all and may include shunts in places but it allows testers and developers to be productive and it gets the entire team thinking at the user level from the very beginning. This is a good practice when building any software system, but is particularly important for services.
- **Test with real data**. Fork user requests or workload from production to test environments. Pick

up production data and put it in test environments. The diverse user population of the product will always be most creative at finding bugs. Clearly, privacy commitments must be maintained so it's vital that this data never leak back out into production.

- **Run system-level acceptance tests**. Tests that run locally provide sanity check that speeds iterative development. To avoid heavy maintenance cost they should still be at system level.
- **Test and develop in full environments**. Set aside hardware to test at interesting scale. Most importantly, use the same data collection and mining techniques used in production on these environments to maximize the investment.

Hardware Selection and Standardization

The usual argument for SKU standardization is that bulk purchases can save considerable money. This is inarguably true. The larger need for hardware standardization is that it allows for faster service deployment and growth. If each service is purchasing their own private infrastructure, then each service has to

1. determine which hardware currently is the best cost/performing option,
2. order the hardware, and
3. do hardware qualification and software deployment once the hardware is installed in the data center.

This usually takes a month and can easily take more.

A better approach is a ''services fabric'' that includes a small number of hardware SKUs and the automatic management and provisioning infrastructure on which all service are run. If more machines are needed for a test cluster, they are requested via a web service and quickly made available. If a small service gets more successful, new resources can be added from the existing pool. This approach ensures two vital principles: 1) all services, even small ones, are using the automatic management and provisioning infrastructure and 2) new services can be tested and deployed much more rapidly.

Best practices for hardware selection include:

- **Use only standard SKUs**. Having a single or small number of SKUs in production allows resources to be moved fluidly between services as needed. The most cost-effective model is to develop a standard service-hosting framework that includes automatic management and provisioning, hardware, and a standard set of shared services. Standard SKUs is a core requirement to achieve this goal.
- **Purchase full racks**. Purchase hardware in fully configured and tested racks or blocks of multiple racks. Racking and stacking costs are inexplicably high in most data centers, so let the system manufacturers do it and wheel in full racks.
- **Write to a hardware abstraction**. Write the service to an abstract hardware description. Rather than fully-exploiting the hardware SKU, the service should neither exploit that SKU nor depend upon detailed knowledge of it. This allows the 2-way, 4-disk SKU to be upgraded over time as better cost/performing systems come available. The SKU should be a virtual description that includes number of CPUs and disks, and a minimum for memory. Finer-grained information about the SKU should not be exploited.
- **Abstract the network and naming**. Abstract the network and naming as far as possible, using DNS and CNAMEs. Always, always use a CNAME. Hardware breaks, comes off lease, and gets repurposed. Never rely on a machine name in any part of the code. A flip of the CNAME in DNS is a lot easier than changing configuration files, or worse yet, production code. If you need to avoid flushing the DNS cache, remember to set Time To Live sufficiently low to ensure that changes are pushed as quickly as needed.

Operations and Capacity Planning

The key to operating services efficiently is to build the system to eliminate the vast majority of operations administrative interactions. The goal should be that a highly-reliable, 24x7 service should be maintained by a small 8x5 operations staff.

However, unusual failures will happen and there will be times when systems or groups of systems can't be brought back on line. Understanding this possibility, automate the procedure to move state off the damaged systems. Relying on operations to update SQL tables by hand or to move data using ad hoc techniques is courting disaster. Mistakes get made in the heat of battle. Anticipate the corrective actions the operations team will need to make, and write and test these procedures up-front. Generally, the development team needs to automate emergency recovery actions and they must test them. Clearly not all failures can be anticipated, but typically a small set of recovery actions can be used to recover from broad classes of failures. Essentially, build and test ''recovery kernels'' that can be used and combined in different ways depending upon the scope and the nature of the disaster.

The recovery scripts need to be tested in production. The general rule is that nothing works if it isn't tested frequently so don't implement anything the team doesn't have the courage to use. If testing in production is too risky, the script isn't ready or safe for use in an emergency. The key point here is that disasters happen and it's amazing how frequently a small disaster becomes a big disaster as a consequence of a recovery step that doesn't work as expected. Anticipate these events and engineer automated actions to get the service back on line without further loss of data or up time.

- **Make the development team responsible**. Amazon is perhaps the most aggressively down this path with their slogan "you built it, you manage it." That position is perhaps slightly stronger than the one we would take, but it's clearly the right general direction. If development is frequently called in the middle of the night, automation is the likely outcome. If operations is frequently called, the usual reaction is to grow the operations team.
- **Soft delete only**. Never delete anything. Just mark it deleted. When new data comes in, record the requests on the way. Keep a rolling two week (or more) history of all changes to help recover from software or administrative errors. If someone makes a mistake and forgets the where clause on a delete statement (it has happened before and it will again), all logical copies of the data are deleted. Neither RAID nor mirroring can protect against this form of error. The ability to recover the data can make the difference between a highly embarrassing issue or a minor, barely noticeable glitch. For those systems already doing off-line backups, this additional record of data coming into the service only needs to be since the last backup. But, being cautious, we recommend going farther back anyway.
- **Track resource allocation**. Understand the costs of additional load for capacity planning. Every service needs to develop some metrics of use such as concurrent users online, user requests per second, or something else appropriate. Whatever the metric, there must be a direct and known correlation between this measure of load and the hardware resources needed. The estimated load number should be fed by the sales and marketing teams and used by the operations team in capacity planning. Different services will have different change velocities and require different ordering cycles. We've worked on services where we updated the marketing forecasts every 90 days, and updated the capacity plan and ordered equipment every 30 days.
- **Make one change at a time**. When in trouble, only apply one change to the environment at a time. This may seem obvious, but we've seen many occasions when multiple changes meant cause and effect could not be correlated.
- **Make Everything Configurable**. Anything that has any chance of needing to be changed in production should be made configurable and tunable in production without a code change. Even if there is no good reason why a value will need to change in production, make it changeable as long as it is easy to do. These knobs shouldn't be changed at will in production, and the system should be thoroughly tested using the configuration that is planned for production. But when a production problem arises, it is always easier, safer, and much faster to make a simple configuration change compared to coding, compiling, testing, and deploying code changes.

Auditing, Monitoring and Alerting

The operations team can't instrument a service in deployment. Make substantial effort during development to ensure that performance data, health data, throughput data, etc. are all produced by every component in the system.

Any time there is a configuration change, the exact change, who did it, and when it was done needs to be logged in the audit log. When production problems begin, the first question to answer is what changes have been made recently. Without a configuration audit trail, the answer is always "nothing" has changed and it's almost always the case that what was forgotten was the change that led to the question.

Alerting is an art. There is a tendency to alert on any event that the developer expects they might find interesting and so version-one services often produce reams of useless alerts which never get looked at. To be effective, each alert has to represent a problem. Otherwise, the operations team will learn to ignore them. We don't know of any magic to get alerting correct other than to interactively tune what conditions drive alerts to ensure that all critical events are alerted and there are not alerts when nothing needs to be done. To get alerting levels correct, two metrics can help and are worth tracking: 1) alerts-to-trouble ticket ratio (with a goal of near one), and 2) number of systems health issues without corresponding alerts (with a goal of near zero).

- **Instrument everything**. Measure every customer interaction or transaction that flows through the system and report anomalies. There is a place for "runners" (synthetic workloads that simulate user interactions with a service in production) but they aren't close to sufficient. Using runners alone, we've seen it take days to even notice a serious problem, since the standard runner workload was continuing to be processed well, and then days more to know why.
- **Data is the most valuable asset**. If the normal operating behavior isn't well-understood, it's hard to respond to what isn't. Lots of data on what is happening in the system needs to be gathered to know it really is working well. Many services have gone through catastrophic failures and only learned of the failure when the phones started ringing.
- **Have a customer view of service**. Perform end-to-end testing. Runners are not enough, but they are needed to ensure the service is fully working. Make sure complex and important paths such as logging in a new user are tested

by the runners. Avoid false positives. If a runner failure isn't considered important, change the test to one that is. Again, once people become accustomed to ignoring data, breakages won't get immediate attention.

- **Instrumentation required for production testing**. In order to safely test in production, complete monitoring and alerting is needed. If a component is failing, it needs to be detected quickly.
- **Latencies are the toughest problem**. Examples are slow I/O and not quite failing but processing slowly. These are hard to find, so instrument carefully to ensure they are detected.
- **Have sufficient production data**. In order to find problems, data has to be available. Build fine grained monitoring in early or it becomes expensive to retrofit later. The most important data that we've relied upon includes:
  - **Use performance counters for all operations**. Record the latency of operations and number of operations per second at the least. The waxing and waning of these values is a huge red flag.
  - **Audit all operations**. Every time somebody does something, especially something significant, log it. This serves two purposes: first, the logs can be mined to find out what sort of things users are doing (in our case, the kind of queries they are doing) and second, it helps in debugging a problem once it is found.

    A related point: this won't do much good if everyone is using the same account to administer the systems. A very bad idea but not all that rare.
  - **Track all fault tolerance mechanisms**. Fault tolerance mechanisms hide failures. Track every time a retry happens, or a piece of data is copied from one place to another, or a machine is rebooted or a service restarted. Know when fault tolerance is hiding little failures so they can be tracked down before they become big failures. We had a 2000-machine service fall slowly to only 400 available over the period of a few days without it being noticed initially.
  - **Track operations against important entities**. Make an ''audit log'' of everything significant that has happened to a particular entity, be it a document or chunk of documents. When running data analysis, it's common to find anomalies in the data. Know where the data came from and what processing it's been through. This is particularly difficult to add later in the project.
  - **Asserts**. Use asserts freely and throughout the product. Collect the resulting logs or crash dumps and investigate them. For systems that run different services in the same process boundary and can't use asserts, write trace records. Whatever the implementation, be able to flag problems and mine frequency of different problems.
  - **Keep historical data**. Historical performance and log data is necessary for trending and problem diagnosis.
- **Configurable logging**. Support configurable logging that can optionally be turned on or off as needed to debug issues. Having to deploy new builds with extra monitoring during a failure is very dangerous.
- **Expose health information for monitoring**. Think about ways to externally monitor the health of the service and make it easy to monitor it in production.
- **Make all reported errors actionable**. Problems will happen. Things will break. If an unrecoverable error in code is detected and logged or reported as an error, the error message should indicate possible causes for the error and suggest ways to correct it. Un-actionable error reports are not useful and over time, they get ignored and real failures will be missed.
- **Enable quick diagnosis of production problems**.
  - **Give enough information to diagnose**. When problems are flagged, give enough information that a person can diagnose it. Otherwise the barrier to entry will be too high and the flags will be ignored. For example, don't just say "10 queries returned no results." Add "and here is the list, and the times they happened."
  - **Chain of evidence**. Make sure that from beginning to end there is a path for developer to diagnose a problem. This is typically done with logs.
  - **Debugging in production**. We prefer a model where the systems are almost never touched by anyone including operations and that debugging is done by snapping the image, dumping the memory, and shipping it out of production. When production debugging is the only option, developers are the best choice. Ensure they are well trained in what is allowed on production servers. Our experience has been that the less frequently systems are touched in production, the happier customers generally are. So we recommend working very hard on not having to touch live systems still in production.
  - **Record all significant actions**. Every time the system does something important, particularly on a network request or modification of data, log what happened. This includes both when a user sends a command and what the system internally does. Having this

record helps immensely in debugging problems. Even more importantly, mining tools can be built that find out useful aggregates, such as, what kind of queries are users doing (i.e., which words, how many words, etc.)

Graceful Degradation and Admission Control

There will be times when DOS attacks or some change in usage patterns causes a sudden workload spike. The service needs be able to degrade gracefully and control admissions. For example, during 9/11 most news services melted down and couldn't provide a usable service to any of the user base. Reliably delivering a subset of the articles would have been a better choice. Two best practices, a "big red switch" and admission control, need to be tailored to each service. But both are powerful and necessary.

- Support a "big red switch." The idea of the "big red switch" originally came from Windows Live Search and it has a lot of power. We've generalized it somewhat in that more transactional services differ from Search in significant ways. But the idea is very powerful and applicable anywhere. Generally, a "big red switch" is a designed and tested action that can be taken when the service is no longer able to meet its SLA, or when that is imminent. Arguably referring to graceful degradation as a "big red switch" is a slightly confusing nomenclature but what is meant is the ability to shed non-critical load in an emergency.

  The concept of a big red switch is to keep the vital processing progressing while shedding or delaying some non-critical workload. By design, this should never happen but it's good to have recourse when it does. Trying to figure these out when the service is on fire is risky. If there is some load that can be queued and processed later, it's a candidate for a big red switch. If it's possible to continue to operate the transaction system while disabling advance querying, that's also a good candidate. The key thing is determining what is minimally required if the system is in trouble, and implementing and testing the option to shut off the non-essential services when that happens. Note that a correct big red switch is reversible. Resetting the switch should be tested to ensure that the full service returns to operation, including all batch jobs and other previously halted non-critical work.
- Control admission. The second important concept is admission control. If the current load cannot be processed on the system, bringing more work load into the system just assures that a larger cross section of the user base is going to get a bad experience. How this gets done is dependent on the system and some can do this more easily than others. As an example, the last service we led processed email. If the system was over-capacity and starting to queue, we were better off not accepting more mail into the system and let it queue at the source. The key reason this made sense, and actually decreased overall service latency, is that as our queues built, we processed more slowly. If we didn't allow the queues to build, throughput would be higher. Another technique is to service premium customers ahead of non-premium customers, or known users ahead of guests, or guests ahead of users if "try and buy" is part of the business model.
- Meter admission. Another incredibly important concept is a modification of the admission control point made above. If the system fails and goes down, be able to bring it back up slowly ensuring that all is well. It must be possible to let just one user in, then let in 10 users/second, and slowly ramp up. It's vital that each service have a fine-grained knob to slowly ramp up usage when coming back on line or recovering from a catastrophic failure. This capability is rarely included in the first release of any service Where a service has clients, there must be a means for the service to inform the client that it's down and when it might be up. This allows the client to continue to operate on local data if applicable, and getting the client to back-off and not pound the service can make it easier to get the service back on line. This also gives an opportunity for the service owners to communicate directly with the user (see below) and control their expectations. Another client-side trick that can be used to prevent them all synchronously hammering the server is to introduce intentional jitter and per-entity automatic backup.

Customer and Press Communication Plan

Systems fail, and there will be times when latency or other issues must be communicated to customers. Communications should be made available through multiple channels in an opt-in basis: RSS, web, instant messages, email, etc. For those services with clients, the ability for the service to communicate with the user through the client can be very useful. The client can be asked to back off until some specific time or for some duration. The client can be asked to run in disconnected, cached mode if supported. The client can show the user the system status and when full functionality is expected to be available again.

Even without a client, if users interact with the system via web pages for example, the system state can still be communicated to them. If users understand what is happening and have a reasonable expectation of when the service will be restored, satisfaction is much higher. There is a natural tendency for service owners to want to hide system issues but, over time, we've become convinced that making information on

the state of the service available to the customer base almost always improves customer satisfaction. Even in no-charge systems, if people know what is happening and when it'll be back, they appear less likely to abandon the service.

Certain types of events will bring press coverage. The service will be much better represented if these scenarios are prepared for in advance. Issues like mass data loss or corruption, security breach, privacy violations, and lengthy service down-times can draw the press. Have a communications plan in place. Know who to call when and how to direct calls. The skeleton of the communications plan should already be drawn up. Each type of disaster should have a plan in place on who to call, when to call them, and how to handle communications.

Customer Self-Provisioning and Self-Help

Customer self-provisioning substantially reduces costs and also increases customer satisfaction. If a customer can go to the web, enter the needed data and just start using the service, they are happier than if they had to waste time in a call processing queue. We've always felt that the major cell phone carriers miss an opportunity to both save and improve customer satisfaction by not allowing self-service for those that don't want to call the customer support group.

Conclusion

Reducing operations costs and improving service reliability for a high scale internet service starts with writing the service to be operations-friendly. In this document we define operations-friendly and summarize best practices in service design, development, deployment, and operation from engineers working on high-scale services.

Acknowledgements

We would like to thank Andrew Cencini (Rackable Systems), Tony Chen (Xbox Live), Filo D'Souza (Exchange Hosted Services & SQL Server), Jawaid Ekram (Exchange Hosted Services & Live Meeting), Matt Gambardella (Rackable Systems), Eliot Gillum (Windows Live Hotmail), Bill Hoffman (Windows Live Storage), John Keiser (Windows Live Search), Anastasios Kasiolas (Windows Live Storage), David Nichols (Windows Live Messenger & Silverlight), Deepak Patil (Windows Live Operations), Todd Roman (Exchange Hosted Services), Achint Srivastava (Windows Live Search), Phil Smoot (Windows Live Hotmail), Yan Leshinsky (Windows Live Search), Mike Ziock (Exchange Hosted Services & Live Meeting), Jim Gray (Microsoft Research), and David Treadwell (Windows Live Platform Services) for background information, points from their experience, and comments on early drafts of this paper. We particularly appreciated the input from Bill Hoffman of the Windows Live Hotmail team and Achint Srivastava and John Keiser, both of the Windows Live Search team.

References

[1] Isard, Michael, "Autopilot: Automatic Data Center Operation," *Operating Systems Review*, April, 2007, http://research.microsoft.com/users/misard/papers/osr2007.pdf .

[2] Patterson, David, *Recovery Oriented Computing*, Berkeley, CA, 2005, http://roc.cs.berkeley.edu/ .

[3] Patterson, David, *Recovery Oriented Computing: A New Research Agenda for a New Century*, February, 2002, http://www.cs.berkeley.edu/~pattrsn/talks/HPCAkeynote.ppt .

[4] Fox, Armando and D. Patterson, "Self-Repairing Computers," *Scientific American*, June, 2003, http://www.sciam.com/article.cfm?articleID=000DAA41-3B4E-1EB7-BDC0809EC588EEDF .

[5] Fox, Armand, *Crash-Only Software*, Stanford, CA, 2004, http://crash.stanford.edu/ .

[6] Hoffman, Bill, *Windows Live Storage Platform*, private communication, 2006.

[7] Shakib, Darren, *Windows Live Search*, private communication, 2004.