**PRIVACY**Privacy regulations, like the European Union’s General Data Protection Regulation (GDPR), the Children's Online Privacy Protection Rule (COPPA), and the California Consumer Privacy Act (CCPA), combined with privacy missteps by technology companies, have raised customers’ privacy awareness and expectations. In a constantly changing regulatory and privacy landscape, customers expect their privacy to be protected, demand transparency, and want control over their data. Amazon needs to ensure that our customers’ privacy rights are always maintained across all of Consumer, Digital, and Other (CDO) so that we retain our customers' trust.

**Key icon Key takeaways**

* Attest in [Kale](https://kale.amazon.dev/) to identify controls your application needs to implement.
* Consult with your [Privacy Bar Raiser](https://w.amazon.com/bin/view/CDO_Privacy/Partner_Teams) and [line lawyer](https://inside.amazon.com/en/services/legal/Pages/LegalContacts.aspx) for privacy guidance as they understand your use case well.
* Business line lawyers will define which data from a dataset needs to be returned in a DSAR.
* PD2 impacts all of a dataset, whereas OD3 impacts data for a particular customer.
* PD2 requires access controls to be implemented for each data subscriber.
* On OD3, data required for Tax, Accounting, Legal (returns, warranties, possible litigation), Fraud, and Finance (TALF) applications will be retained, and all other customer data will be deleted. Access to this data needs to be limited to TALF applications.

**Privacy tenets**

The S-Team has defined a set of tenets to adhere to as you consider the collection, use, and sharing of customers' personal data. These tenets are intended for use across Amazon, so they are broadly defined, but you can collaborate with your team to develop your own more detailed guidelines. The following privacy tenets are the most important for you to keep in mind:

1. We endeavor to make our collection, use, or sharing of personal data intuitive to the customer.  We also disclose to customers what personal data we collect, how we use it, and whether we share it.  We do not want customers to be surprised by how we collect, use, or share their personal data.
2. We strive to design products and services that give customers control over whether and how their personal data is collected, used, or shared.  These choices include giving customers the ability to elect whether to use a product or service, to opt into or out of the collection or use of certain personal data, to access and update certain personal data, and to delete certain personal data.
3. We endeavor to limit collection of personal data to what is necessary for the provision or improvement of the products and services for which it was collected.  When doing so is consistent with accomplishing those purposes, we seek to de-identify or pseudonymize personal data to minimize its retention and use.
4. We do not sell our customers’ personal data.  When we share personal data with third parties, in the limited circumstances described in our privacy notice, we endeavor to share the least amount necessary in order to accomplish the desired result.
5. We strive to design, build, audit, and continually improve our products, services, and back-end systems to enable us to meet these tenets.

For the complete list, see [Amazon Tenets for Collection, Use, and Sharing of our Customers’ Personal Data (Unless you know better ones....)](https://w.amazon.com/bin/view/CDOPrivacy/AmazonPrivacyTenets/).

**What is personal data?**

**Personal data** is any information relating to a person who could be identified, directly or indirectly, by an identifier within that information. For example, a name, an identification number, and location data. Amazon treats any information linked to a customer account as personal data that we must keep private to protect the customer privacy.

For the current list of personal data identifiers, see [Data Element Registry](http://der.data-governance.privacy.a2z.com/). You can learn more about how the registry is developed and maintained in the [CDO Privacy Engineering wiki](https://w.amazon.com/bin/view/PrivacyEngineering/Product/DER).

For more detailed information about customer privacy, see [CDOPrivacy FAQs](https://w.amazon.com/bin/view/CDOPrivacy/FAQs/).

**Privacy Controls for 2021**

The CDO Privacy (CDOP) organization will support the following programs and tools in 2021:

* **Data Subject Access Request (DSAR):** Applicable to all authoritative datasets where *authoritative dataset* is defined as the first collection and storage point of customer data in Amazon. If a customer submits a DSAR request, we return all related customer data that is held in the authoritative datasets. To avoid returning multiple copies of the same data to a customer, only authoritative datasets need to respond to a DSAR request. Authoritative data must be returned in a customer-friendly manner that is easy for the customer to understand. For example, provide the name of the author or book said customer ordered, instead of the associated ASIN number.
  + The CDOP tool AEDU (Amazon Electronic Documents Upload) receives customer requests for DSARs. *ALL AUTHORITATIVE DATASETS* need to onboard and return customer personal data to AEDU. To learn more, see [AEDU Onboarding](https://w.amazon.com/bin/view/CDOPrivacy/Onboarding/DSAR/AEDU/).
* **On Demand Data Deletion (OD3):** On customer request, all copies of data need to be deleted except where a service has legal reason to continue to use the data. Legal reasons are limited to Tax, Accounting, Fraud, Sanctions, Returns, Warranties, and Litigation or possible Litigation.
  + The CDOP tool Sandfire receives customer requests for OD3. *ALL DATASETS THAT CONTAIN CUSTOMER PERSONAL DATA* need to onboard to Sandfire so that, on customer request, they can delete customer personal data that we have no legal reason to retain. To learn more about data deletion onboarding, start with the [CDO Privacy Deletion Checklist](https://w.amazon.com/bin/view/CDOPrivacy/DeletionGuidance/DeletionChecklist).
* **Programmatic Data Deletion (PD2):** Triggers data deletion automatically, when all approved use cases as outlined in the [Data Retention and Deletion Standard (DRDS)](https://policy.amazon.com/standard/50725) expire.
  + Because this is programmatic deletion, *ALL DATASETS THAT CONTAIN CUSTOMER PERSONAL DATA* need to automatically delete personal data as retention expires.
* **Privacy review:** Every service owner must attest in [Kale](https://kale.data-governance.privacy.a2z.com) whether their service stores or processes customer personal data or not. If it does, the owner must further attest to all personal data elements collected or consumed, and identify their intended business use.
  + *ALL SERVICE OWNERS,* irrespective of personal data, need to attest in Kale so that they are informed of all the privacy controls required.
  + Legal review and approval of Kale attestations can take up to 6 weeks, and implementing required controls may require multiple months of effort.
* **Training:** All blue badge employees will receive basic privacy training, and SDEs/SDMs/TPMs/PMTs will receive role-based training. Training will be automatically assigned to you.

**Example**

The following diagrams show the PD2 and OD3 process timelines for a data store that contains personal data with three business use cases: Marketing and Advertising, Personalizing the Customer Experience, and Tax.

**PD2 (Applies to complete dataset)**  


In Programmatic Data Deletion (PD2), the retention period for marketing and advertising data expires first. Access to data is then limited to other uses. The retention period for customer experience data expires next, and data access is limited to tax purposes only. The retention period for tax-related data expires last, and the personal data is deleted.

**OD3 (Applies only to requesting customer)**  


In On Demand Data Deletion (OD3), data retained for marketing and advertising purposes is deleted upon customer request, and Amazon's access to data is limited to tax-related data only. OD3 can apply at any time during the standard data retention periods shown in the PD2 diagram. If the customer requests that their data be deleted, only the tax-related data is retained until its standard retention period expires.

**Privacy steps for deployed applications and data stores**

1. As in the privacy review practice mentioned in the previous section, every service owner must attest in Kale so that they are informed of all privacy controls required.
2. Based on Kale analysis, no further action may be needed...OR applications may need to onboard to Sandfire and AEDU, and implement PD2 within 6 months.
3. If you need to onboard to Sandfire and PD2, do the following:
   1. Submit dataset to Tax, Accounting, Legal, Fraud, and Finance to determine what data can be retained and what data needs to be deleted.
   2. Run a campaign to understand all one level downstream client use cases.
   3. Enable a mechanism for new subscribers to the data to first register their use case.
   4. Implement fine grain access control, data deletion, and column level control, if needed.

**Privacy steps for new applications by Software Development Life Cycle (SDLC) phase**

The following phases of application development have specific privacy considerations for each.

**Ideation phase**

It is important to consider Privacy at the beginning of the application's development cycle and to include the relevant direction in the PR/FAQ. Addressing privacy issues after a product launch could have a negative impact on customer trust, cost Amazon financial loss due to rework, project delays, and compliance fines. Principal Engineers (PE), Privacy Bar Raisers, line lawyers, and CDOP can help you answer the following questions:

1. What is the specific business purpose of this application or service? What personal data do you need to achieve the business purpose?
2. Has the customer provided consent for their data to be used for that business purpose? Could you use data minimization techniques to meet the business purpose?
3. Can fully aggregated, anonymized, or pseudonymized data meet the business purpose?
4. What impact might a data leakage in this application have on Amazon or Amazon customers?

**Planning and design phase**

In the planning and design phase, you should focus on minimizing the personal data you collect. You need to include relevant privacy controls in the overall design for the personal data you process, collect, store, or share. Remember, PEs, Privacy Bar Raisers, line lawyers, and CDOP are here to help! Keep the following principles in mind:

1. [Data minimization](https://w.amazon.com/bin/view/CDOPrivacy/DeletionGuidance/Techniques) is one of the core constructs of privacy regulations. If you do not collect personal data, you are not in scope for privacy controls.
2. Attest in Kale to identify which privacy controls apply to your service, and inform Kale of the bindles your service or datastore will use. Do you need to onboard to AEDU or Sandfire? Do you need to implement PD2?
3. If you do need some personal data, is it already collected by some other team or do you need to collect it? Do you need to capture consent for your use case, or did the customer provide consent when they opened their account?
4. If you share personal data, how will you capture downstream clients' use cases? How will you implement fine-grain access control so that each downstream client only receives data as permitted by [Data Retention and Deletion Standards (DRDS)](https://policy.amazon.com/standard/50725)?
5. If you receive data from upstream systems, what is the subscription mechanism that ensures your service receives all data as permitted by [Data Retention and Deletion Standards (DRDS)](https://policy.amazon.com/standard/50725)?
6. Use CloudAuth for new services, or AAA for legacy services, to meet the requirements for API authentication and authorization. These services ensure that transport layer security is in place to secure data in transit.
7. Make sure that personal data does not end up in unintended locations, like logs, audit records, or error messages.
8. Do you share the data you are collecting with anyone outside Amazon in order to achieve your business purpose? Make sure that you are following appropriate [third party security approval](https://w.amazon.com/bin/view/ThirdPartySecurity/) processes, as outlined by CDO SRC Third Party Security. Adequate safeguards, such as sharing protocols or agreements with the third party, must be in place.
9. Do you use cookies and cater to the EU marketplace? You will need to adhere to [EU affirmative cookie consent](https://w.amazon.com/bin/view/EU_Cookie) requirements.

**Implementation phase**

Continue attesting to any changes in Kale as you go through the implementation phase, and ensure your business line lawyer is reviewing the changes. If you identify new controls to be met, go back to the planning and design phase. Implement all controls that need to be implemented, and consider the following areas during your implementation phase.

**Flexibility**

New regulations on privacy are being introduced across countries. This might result in new controls or changes in retention and deletion standards.

* Do not hard-code retention or deletion time frames. Make sure that these time frames are easily modifiable configurations.
* Today, consumer privacy requirements are defined based on customer profiles, but there could be multiple individuals using the same customer profile. New controls might require you to provide privacy using [Person Centric Amazon (PCA) profiles](https://w.amazon.com/bin/view/IdentityServices/PersonCentric/).

**Schema design**

Name column headers as identified in the [Data Element Registry](https://der.data-governance.privacy.a2z.com/) as you finalize your schema. This will provide a better customer experience during Data Subject Access Requests (DSAR), and allow CDO Privacy tools to find personal data easily in data stores.

**Logs**

Ensure that the logs (audit, debug, and verbose) you collect include no or minimal personal data.

**Permissions and access control**

Use Bindles and Cloudauth/AAA to manage access control at a granular level.

**Testing phase**

Verify the following in positive and negative test cases:

* Customers have the ability to review (DSAR) and delete personal data
* Customers can withdraw or refuse consent
* Parents/guardians can provide, withdraw, and refuse consent for children (PCA)
* Personal data can be imported and exported to support data portability
* Data is stored in correct locations and platforms, and only in those locations and platforms
* Access to authorized data is possible for the customer, and for others that have varying retention windows
* Exercise the third party notification mechanism, if applicable

**After launch and/or maintenance phase**

Re-attest in Kale on a yearly basis, or on any change, to identify new controls that need to be met.

**SECURITY**

**Why security is important**

Security is foundational to everything that we do at Amazon. Without security ingrained into our systems and processes, we would put customer trust at risk. The Consumer Security [tenets](https://w.amazon.com/bin/view/Infosec/Tenets/) for CDO (Consumer, Digital, and Other) include: *Customer privacy, security, availability, customer experience, and cost efficiency, in that order, earn customer trust.*

When required, we prioritize privacy and security over availability, customer experience, and cost efficiency. At Amazon, nothing is more important than maintaining the trust that our customers place in us.

**Shared responsibilities**

Key icon**Key takeaway:** Both you, as the service owner, and the security teams in Consumer have a role to play in securing your service. You should understand what parts you are responsible for and what the security teams are responsible for. If something is not clear, please ask the security team.

Security is a partnership between you, your team, and the various security teams that operate throughout Consumer, Digital, and Other (CDO). We’ll cover more about the responsibility model we use later on, but we consider security a “shared responsibility” with all participants contributing. Security is a component of any well-architected application, just as [reliability](https://w.amazon.com/bin/view/CEH/Availability/), [efficiency](https://w.amazon.com/bin/view/CEH/Efficiency/), and [operational excellence](https://w.amazon.com/bin/view/EE/Learn/OE/) are required to build robust applications.

CDO security teams do three main things:

* Provide guidance on how to build secure applications
* Create tools that help service owners to build and operate securely
* Continually audit at a Consumer-wide level to ensure that we're all raising the bar

All of this is intended to deliver against our most important mission, maintaining customer trust.

We understand that no two systems are the same, that every team faces unique challenges throughout the entire lifecycle of a system, and that teams do need assistance from time to time. Therefore, we have experts available to provide guidance and input on how to solve issues that arise. Saying “No” is the last resort for us, and we want to work with each team to find solutions to their problems, to enable successful and on-time launches while maintaining the right level of security.

If you read no further in this section, please take away these points:

* Security is here to support you and reduce the work you need to do in order to build secure systems.
* Security at Amazon is not about blocking you but about finding ways to accomplish your goals in a secure manner.
* If in doubt, [come and talk to us](https://w.amazon.com/index.php/Infosec/Amazon_Information_Security). If it’s not clear why things are done a certain way, ask us!

For much more information about security's place in well-architected applications, see [Security Pillar - AWS Well-Architected Framework](https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html).

**How security teams think about security**

To develop and operate a secure service at Amazon, without an associated high cost, you need to understand how we think about security at Amazon. In this section, we will review our mental model for security, what security tools exist, and how to stay secure with minimal effort during the lifecycle of the service being built.

**Note:** Throughout this section of the CEH, we will use the generic “security team”.  With the scale and breadth that CDO covers, there are multiple security teams within CDO. [Consumer Security](https://w.amazon.com/bin/view/Infosec/) is responsible for the security of [CDO](https://w.amazon.com/bin/view/CDO/), including all CDO-owned subsidiaries. The [BST (Business Security Teams)](https://w.amazon.com/bin/view/InfoSec_Engagement_Team/FederatedSecurityTeam/) model places security teams in organizations that require domain-specific, differentiated security expertise. If you are in a development team in scope for a BST, you will see both Consumer Security and your BST engaging with you. Internally, we all work together to ensure that there is consistency in guidance and tooling and that there is the minimum amount of duplication created. If you ever experience conflicting guidance, please reach out to your BST and/or Consumer Security and we will provide the required clarity. The intent of our model is to ensure that differentiated security work occurs in the BSTs, and undifferentiated work occurs in Consumer Security.

A service owner is ultimately responsible for the security of their service...but we should not mistake this for a service owner having to do all of the work required to build and operate a secure service. A service owner needs to understand how their service is secured, that they have chosen the right technologies, and that their day-to-day operations are leveraging current known best practices. However, vetting which platforms are secure, how to write robust code, or creating those operational best practices are not the responsibilities of a service owner.

The security teams across Consumer (CDO InfoSec, and where present, the Federated Security Teams) exist to provide the differentiated security foundation for service owners to build on. When we have problems that exist across the whole of Consumer, it is the security teams’ responsibility to understand that need and produce a solution that can be widely adopted. This might include code scanning, threat detection, or providing guidance on the right version of cryptography to use.

Guidance is provided in various formats. For example, we write policy and guidance documents, but also provide [AWS CloudFormation templates and CDK Constructs](https://highcastle.a2z.com/) that make it easy to implement each policy. A service owner is responsible for picking the right option, balanced against their unique needs.

As a central security team looks for anomalies at scale, what is normal for everyone else might be an exception for your service. As we look for misconfigurations, your configuration might be appropriate for the business case you are solving. This is where the service owner has to play an active role in understanding how much the required security capabilities are provided to them and how much they need to deliver themselves.

Color-coded flow chart of the 8 steps of security best practices, as described in the following sections.

As shown in the previous diagram, the 8 steps of security best practices are associated with different stages of development. Steps 1-3 are part of the Design stage. Step 4 is part of the Build stage. Steps 5 and 6 are part of the Deploy stage. Step 7 is part of the Operate stage. Step 8 is part of the Evolve phase. All 8 steps are described in the following sections.

**Important:** As a service owner, you are free to pick the right technologies and processes to run your business. When picking a language to write a new service in, you can choose everything from Java to Erlang. But each one has benefits and drawbacks that need to be understood. Java is a well-supported language at Amazon, both from a tooling perspective and within the development community. Our supported frameworks in Java (such as Java/Horizonte) have part of our standard security policies baked in. But if using a different language is the right thing for your service, and you understand what you are giving up by picking it, you are free to choose it. Just understand that if you decide to use something less supported (such as Erlang), you might have more work to do to secure your system.

**Step 1: Understand Amazon's security policies and standards**

Key icon**Key takeaway:** Amazon policies and standards are documented in [policy.amazon.com](https://policy.amazon.com/) and the [Security Knowledge Base](https://skb.corp.amazon.com/). Read the [Data Classification Catalog](https://policy.amazon.com/policy/97) and [Data Handling Standard](https://policy.amazon.com/standard/99) policies first to better understand how to architect your systems and match the security against the data you are processing.

To provide clarity for builders, Consumer Security publishes governance policy in [policy.amazon.com](https://policy.amazon.com/) and  [skb.corp.amazon.com](https://skb.corp.amazon.com/). These resources provide clear guidance on how our systems need to be built and operated in order to provide a secure environment for handling the data that our customers entrust us with. Some of the policies and governance that we have are related to external compliance standards, such as PCI/DSS (Payment Card Industry, Data Classification Standard) for payment instrument handling or HIPAA (Health Insurance Portability and Accountability Act) for healthcare data handling.

Just like all other teams at Amazon, security wants to automate as much as possible. For example, we have the [ACF (Amazon Control Framework)](https://w.amazon.com/bin/view/InfoSecSRC/SecurityAssurance/AmazonControlFramework/), which provides clarity on what security controls are required, how they are performing, and where they are missing. It also provides a consistent method for scoping, testing, and remediating those controls and failures. Security also works with BT (Builder Tools) to provide tools such as [Quilt](https://w.amazon.com/bin/view/Quilt/), which provides solutions for Operating System (OS) patching and keeps your instances patched with service level agreement (SLA), without causing impact to your fleets.

If the policy doesn't provide clear guidance, each policy page has a **Contact Us** button at the top that you can use to get clarity from our policy team. If you are unable to meet the policy, there is also a **Request Exception** button on each policy page. If you know you are unable to meet the policy, use that button to engage with us and we will work with you to ensure that the exception is recorded and accepted by the appropriate leadership. As previously noted, the intent of security is not to get in the way of your development and build activities, but to ensure that the right auditing mechanisms are put in place to ensure an acceptable level of risk is accepted and recorded.

You should familiarize yourself with all of the policies that are in the [Consumer Security folder](https://policy.amazon.com/#folder=130&types=guideline.other.policy.procedure.standard&ignore_home=true) in policy.amazon.com. For all builders, we recommend you read the following documents.

**Note:** The following links point to the English language versions of our policies. Translated documents are also presented in the [policy folder](https://policy.amazon.com/#folder=130&types=guideline.other.policy.procedure.standard&ignore_home=true).

* [Data Classification Catalog](https://policy.amazon.com/policy/97)
* [Data Handling Standard](https://policy.amazon.com/standard/99)
* [AWS Usage Standard](https://policy.amazon.com/standard/1150)
* [Third Party Security Assessment Policy](https://policy.amazon.com/policy/107)
* [Vulnerability Management Policy](https://policy.amazon.com/policy/141)

**Step 2: Begin the security review process**

Key icon**Key takeaway:** The earlier you begin the review process, the better and easier your end state will be. Skipping or deferring engagement with your security team could cause delays and disruption to your launch.

Consumer Security provides tooling that guides you through the review process. This ensures that the appropriate artifacts are collected, the design is appropriately reviewed, and documents, such as the incident response plan, are created to aid in security events that will occur during the lifecycle of your application. Capturing all of this information also allows your [security certifier](https://w.amazon.com/bin/view/Infosec/Security_Certifier_Program/), if your application requires one, to review and evaluate the security decisions being made in the design.

The [Anvil](https://anvil.amazon.com/) tool guides you through the security review process. It provides a repository for all of your security design artifacts, penetration testing reports, and incident response plans. If risks are identified during the review process, they will be captured in Anvil, which also provides a workflow for risk acceptance by your business leaders.

Having an up-to-date and accurate inventory of all applications at Amazon helps Consumer Security during incident response. For example, an incident response plan might not be needed until years after an application launches, and with none of the original development team executing it. Having a single place for both the owners of the application and the security response team to retrieve the incident plan ensures that everyone is executing from the same runbooks. This facilitates the right level of collaboration during an incident. Like commenting code, creating accurate documentation about the security intent and state of your application is paying it forward for future maintainers.

For some of our services, we need to send data to third parties (non-Amazon companies or AWS) or integrate with third parties. Consumer Security provides the [TPS (Third Party Security)](https://w.amazon.com/bin/view/ThirdPartySecurity/) program to help you audit any third parties that you need to communicate and share information with. As Amazon is the custodian and trusted party for our customer data, we need to ensure that our third-party partners are adhering to the same level of customer obsession when it comes to data protection. The SRC (Security, Risk and Compliance) team is there to help provide the right audit of third parties.

**More resources**

* [Anvil getting started guide](https://w.amazon.com/bin/view/Anvil/CustomerWiki/userguide/)
* [Security Certifier program](https://w.amazon.com/bin/view/Infosec/Security_Certifier_Program/)
* [Anvil application classification documentation](https://w.amazon.com/bin/view/Anvil/CustomerWiki/userguide/#HStep3-FillouttheClassificationSurvey)

**Step 3: Document a threat model**

Key icon**Key takeaway:** A threat model is one approach to enumerating all of the circumstances or events that are harmful to system operations, and the mitigation controls needed to defend against these issues. Doing this before you begin writing code will help with the security design of your system.

The best time to invest in threat modeling is during design, and on an ongoing basis whenever design changes are being considered. Design changes evolve over time, so threat modeling at Amazon is a continuous process. Here are some cases when threat modeling is recommended:

* Your team is designing a new system.
* The existing threat modeling is obsolete.
* You need to make a change to data classification, system design, a dependent component or API, or the details of a data flow.

Objectives of threat modeling:

* **Understand the threats.** Make sure you consider all the adversary’s potential threats for your system, even those that you think have been dealt with. You can track them in a list, or a table.
* **Discover vulnerabilities.** Threat modeling identifies vulnerabilities (unmitigated threats). This helps the application owner understand the risks their app might introduce to Amazon and to implement countermeasures.
* **Document the model.** Threat modeling produces a threat model document, which should evolve as the application changes. This document can be created using [Design Inspector](https://design-inspector.a2z.com/) or the [Anvil](https://anvil.amazon.com/) templates.
* **Provide a basis for secure design and implementation.** Based on the list of threats, the development team knows the security features that need to be implemented and later verified through code reviews and security testing.
* **Provide inputs for the next phases of security review.** A threat model identifies the security-critical parts of your system and helps identify controls that need to be verified through code reviews and security testing.

**More resources**

* [Threat Modelling at Amazon](https://w.amazon.com/bin/view/Threat_Model/)
* [EC2 Threat model from SKB](https://skb.highcastle.a2z.com/implementations/185)
* [Examples of good threat models](https://w.amazon.com/index.php/AWS_IT_Security/Security_Reviews/Threat_Modeling#Can_you_point_me_to_examples_of_good_threat_models.3F)

**Step 4: Build a secure system**

Key icon**Key takeaway:** At every level in the application stack, there are solutions provided for you that already have security baked into them, reducing the work you need to do in order to operate your service securely.

Consumer Security intends that all services operate in a “secure-by-default” mode. In other words, you as the service owner should be able to easily pick the right security settings, and maintain those settings as the security threat landscape changes.

When choosing the technologies to build their system, every service owner has almost unlimited possibilities, but each one has an associated cost. If you pick a technology that no one else at Amazon is currently using, for example, then you will take on the sole cost of ensuring that it meets the security bar required for the data you are handling. By using more well-supported technologies, you will inherit the secure-by-default configurations and tooling to operate your instance securely. Whether you are picking the coding language, the compute run-time, or the database, the same principles apply.

We also recognize that not every application is built the same or uses the same technologies or platforms. For example, if you are developing for [mobile devices](https://w.amazon.com/bin/view/Infosec/Mobile_Security/), [Internet-facing services](https://w.amazon.com/bin/view/Infosec/EdgeSecurity/) or [cloud computing](https://w.amazon.com/bin/view/Infosec/CloudSecurity/), your environment might have unique challenges. We have SMEs available to assist you in tailoring your design for the challenges of each.

Here are some of the tools and solutions that are provided to assist you in running your service:

* [Conduit](https://w.amazon.com/bin/view/ConsumerCloudTrust/Conduit/) - Required to manage and access your AWS accounts.
* [HEX-AWS](https://w.amazon.com/index.php/HEX/HEX-AWS) - Managed hosting in AWS.
* [Honeycomb](https://w.amazon.com/bin/view/Honeycomb/) - Serverless Compute Platform.
* [AppSec Scanners](https://w.amazon.com/bin/view/InfoSec/SDL/Scanners/) - Code scanners project.
* [Security Knowledge Base (SKB)](https://skb.highcastle.a2z.com/) - Guidance and recommendations for secure operation of services.
* [High Castle](https://highcastle.a2z.com/#/)- Marketplace for CDO-approved blueprints.
* [Teams](https://permissions.amazon.com/a/team) - Create groups based on attributes.
* [AccessReview](https://review.access.a2z.com/) - Required to review access to your resources and baseline them.
* [Tokenator](https://w.amazon.com/bin/view/Tokenator) - A key/value datastore for secret data.
* [CloudAuth](https://w.amazon.com/bin/view/Dev.CDO/UnifiedAuth/CloudAuth/) - A solution for CDO system Authorization and Authentication needs.
* [Bindles](https://w.amazon.com/bin/view/Bindles/) - Supports strong ownership for a collection of resources that should be managed as a single entity.
* [Secrets](https://w.amazon.com/bin/view/Secrets) - Covers which solutions are appropriate for managing secret information.
* [Redfort](https://redfort.amazon.com/) - Manages your TLS certificates.
* [SIEGE](https://w.amazon.com/bin/view/SIEGE/) - Protects your data as it moves between intermediate systems.

**Step 5: Complete the security review process before launching your system**

Key icon**Key takeaway:** Completion of a security review does not mean fixing every security issue; it means that you understand the security state that you intend to launch with, and have everything documented and risk-accepted, if required.

As you progress through your application security review, you will find security issues that might not be considered launch blockers. Working with your [Security Certifier](https://w.amazon.com/bin/view/Security_Certifier/) and your leadership team, you can dive deep into each identified issue and decide whether it’s something that needs to be fixed before launch or can wait.

There is also a middle ground between implementing the fix and accepting the risk of launching without the identified problem being fixed. You can explore how to build mitigating controls or defense-in-depth. For every issue identified, you can analyze the path in your workflows to get to that issue and identify how to prevent the issue or bug from being triggered. Alternatively, you can identify where other controls within your system would be able to provide protection.

For certain classes of applications, you will be required to have your application undergo a penetration test. Our [penetration testing team](https://w.amazon.com/bin/view/Infosec/Threat_Analysis/Pentest/Home/) is on hand to guide you through the criteria to see if a test is required and to get it scheduled at the right time in your development cycle.

**Step 6: Deploy your secure system**

Key icon**Key takeaway:** Failure to keep your service in compliance with the required security bar (for example, with completed certification or patching) may result in your deployments being blocked.

While we have talked about guidance and risk acceptance in this section, there are also times when an application gets into an unacceptable state. This might be because the application has not completed its certification, because the list of human or system actors with access is now out of date, or because vulnerable software has been included in the environment. When this state occurs, one of the mechanisms used to get an application back into compliance is to suspend its deployment and redeployment capabilities.  Security integrates with Amazon’s build and deployment systems to trigger this state, using a service called [Security Deployment Controls](https://w.amazon.com/bin/view/InfoSec/Security_Deployment_Controls/).

To facilitate patching or emergency deployments, there is also a mechanism to accept risk and override the block once. This is the [Keys](https://w.amazon.com/bin/view/Infosec/Security_Deployment_Controls/Keys) tool, which allows you to unlock the deployment if you are unable to fix the issue that got your system into the blocked state in the first place.

Blocking deployments is a necessary final step to allow us to bring applications back into compliance, but if you are following the guidance described throughout this section, especially per using well-supported frameworks or [Quilt](https://w.amazon.com/bin/view/Quilt/), then you should never encounter software deployment blockages.

**Step 7: Resolve any operational security issues that are brought to your attention**

Key icon**Key takeaway:** Consumer Security will detect issues and surface those that require timely action or risk acceptance.

Consumer Security scans the CDO environment, checking configuration state data and running risk and threat analyses to find unwanted states. When we find such a state, we surface it through various mechanisms, such as [Shepherd](https://shepherd.a2z.com/). In certain circumstances, we’ll use the [CTI](https://cti.amazon.com/) associated with the owning [Bindle](https://bindles.amazon.com/) to notify you of action that you need to take. So, make sure your ownership information is accurate; it’s a key part of keeping your service secure and getting timely notifications for any action that must be taken.

Issues we discover during our scanning and detection processes are assigned a severity and an associated remediation SLA. Each issue is unique; some are low severity and can be addressed at a later point, while some are severe enough to require immediate action--for example, when there is observed exploitation of a vulnerability, or when the configuration state could allow unwanted access to data held within a system.

When dealing with the breadth of CDO, there are going to be cases when the default evaluation and rating of a vulnerability does not align with your unique environment. For example, this can occur when dealing with physical devices, such as robots or drones, which have additional safety requirements. For these cases, you can reach out to us to reassess the specifics of the vulnerability and adjust the cost to reflect the uniqueness of your use case.

**More resources**

* [PolicyEngine](https://policyengine.amazon.com/) dashboard
* [Shepherd](https://shepherd.a2z.com/)
* [AVP - Amazon Vulnerability Portal](https://vulnerabilities.amazon.com/)
* [Software Assurance Services](https://sas.corp.amazon.com/)
* [Vulnerability Assessment and Management Standard patching SLAs](https://policy.amazon.com/standard/25843)

**Step 8: Continuously iterate to improve your system’s security posture**

Key icon**Key takeaway:** Security of a deployed service is an ongoing process. Any service that is not getting care and attention will become insecure over time.

The security of any system degrades over time. New systems needing access are launched, old systems are deprecated, and changes to organizational structure change access requirements. New vulnerabilities and previously unknown attacks are found. The requirements and assumptions that your system is based upon will change over time. For example, you might have built a system to handle confidential data, but you are now being asked to handle critical data.

As you plan your sprints and develop yearly plans, you will need to factor in some time for maintenance of the security state of your system. Like maintaining a code base, incremental and sustained investment will ensure that there is minimal disruption to your planning and deliverables. Security teams, when absolutely necessary, will run campaigns using the services listed in this section to drive a high bar across the Amazon environment...but this is always the last resort. We prefer to bake in security and provide 0-click or 1-click options for improvements that can be rolled out in a continuous and sustained fashion.

A security campaign is a request for work from some, or all, Amazon development teams.  It can be triggered by the need to globally raise the bar on encryption (for example, deprecating the use of TLS1.0/TLS1.1), a vulnerability (such as [Shellshock](https://en.wikipedia.org/wiki/Shellshock_%28software_bug%29) or [Heartbleed](https://en.wikipedia.org/wiki/Heartbleed)), or to assist in the deprecation of commonly used tools that are about to reach end of life (for example, moving from Amazon Linux 1 to Amazon Linux 2). For some efforts, where there is a well-understood end date (Operating Systems EOL dates), campaigns will start as a notification in our tooling, or with a low-priority trouble ticket. As the end date gets closer, we will gradually increase to sev2.5 or sev2 tickets as required to complete the campaign on time. For events such as vulnerability disclosure and active exploitation, we may need to go straight to high-severity, sev2 tickets. The goal of each campaign is to raise the bar for security, minimize the work effort and disruption for service owners, and still ensure that we do not have unwanted security gaps in our environment.

**Finally**

Security is here to help guide you through the lifecycle of your service. Cradle to grave, security’s job is to provide solutions that help reduce the work effort of keeping your service secure against known threats.

For anything that you need in the future, visit <https://security.a2z.com/> and engage with us. We are here to help!

**AVAILABILITY**

## Availability and resilience

We aim to be ready and able to serve our customers whenever they want or expect our services, delighting them with a level of availability that is "indistinguishable from perfect."

But failures happen. In fact, they are always occurring, from frequent minor failures like dropped network packets to rarer major failures like Availability Zone power outages. To achieve a high level of availability, we design our systems and development processes to keep those failures from impacting our customers. Resilience is our ability to handle failures in such a way that our customers' experience is as close as possible to normal. To ensure resilience, we [design our systems to tolerate failures automatically](https://w.amazon.com/bin/view/EE/Learn/OE/) (see *Improve with metrics*), and define robust operational practices to [detect, mitigate, and resolve failures](https://w.amazon.com/bin/view/EE/Learn/OE/OncallEventResponse/) that are not handled by automation. Then we regularly test that these designs and operational practices actually work to handle the failures.  When our actual performance does not meet our customer-obsessed expectations, we [analyze the failures and define corrective actions to improve](https://w.amazon.com/bin/view/NewCOE/UserGuide/).

The remainder of this page provides a high-level introduction, along with pointers to much more material that will help you to understand and improve the availability and resilience of services under your care.

The guide [Building Reliable and Resilient Services](https://w.amazon.com/bin/view/Services/) is a good source of more information on these topics.

## How we measure availability

We have different ways of measuring availability for an entire site (for example, amazon.com or amazon.de) and for individual services that comprise the site.

We measure availability at the site level as the number of minutes that we are not in an "outage" condition relative to the total number of minutes in a given period:  
availability=1−outage minutestotal minutes  
In this formula, an "outage minute" means a minute in which the condition of the site impacts the ability of our customers to make purchases. Our primary indicator of an outage condition is a drop in the [retail order rate](https://orderratedata.amazon.com/) of the site relative to a predicted value; this is called an "order drop."  We generally express the availability metric as a percentage. The Consumer organization maintains goals on this metric in each region, and [records each recognized retail order drop event](https://outages.amazon.com) to [track our status on these goals](https://w.amazon.com/bin/view/AvailabilityProgram/2020GoalStatus/).

For individual services, we usually measure availability as the service's rate of success at processing units of work in any given period of time. Units of work can be valid incoming requests, tasks, or events for task- or event-based services. For example, for a request-based service, this would be:  
availability=successful responses processedvalid requests received  
This metric is meaningful in short time intervals, even a single minute or less. It can also be specialized to different subsets of requests, like the requests for different APIs or types of work, or the requests from different clients. This makes this metric suitable for dashboards and monitoring, and we expect service teams to monitor their availability metrics against their targets. Working with their customers, service owners set their own targets, known as service level agreements (SLAs), for their services. We usually express the success ratio as a percentage, and typical Tier-1 services (as defined in the next section) target keeping their success rate availability in the 99.99% to 99.999% ("four to five 9s") range in each minute.

As with the "outage minute" for the overall site, one can also define a "service outage minute" when the service's success rate availability metric in that minute falls below a target SLA, or when the average of some percentile of its latency metric in the minute goes above a target SLA. One can then compute an availability metric based on service outage minutes, just as we do for outage minutes on the overall site. This form of availability metric is more appropriate to longer-term team goal setting, such as annual team goals for service availability.

If you're implementing your service availability metric, please read our [guidance on measuring service availability and the relationship to latency](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance#HAvailabilityMetricMeasurement).

### More resources

* [Wikipedia's definition of Availability](https://en.wikipedia.org/wiki/Availability) (similar to our outage minute measure)
* [Availability Best Practices](https://w.amazon.com/index.php/LivePipelineTemplates/BestPractices)
* [AWS Great Availability Expectations](https://w.amazon.com/index.php/LivePipelineTemplates/AwsGreatAvailabilityExpectations)

## SEV-1 Impact, Tier-1 Services, Large Scale Events, and TechOps

When we do have problems, we classify their impacts numerically by severity, from SEV-5 (least severe, productivity not affected) to SEV-1 (most severe, business-critical function down).  For each impact, there are different expectations for the response and interactions provided by service owners. We use Trouble Tickets (TTs) in [tt.amazon.com](https://tt.amazon.com) (deprecated) and Simple Issue Manager Tickets (SIM-T) in [t.corp.amazon.com](https://t.corp.amazon.com/) (new) to track and coordinate the response to high-severity events. For all SEV-1 and some SEV-2 events, we also use conference calls to improve interactions.

Large Scale Events (LSEs) are events impacting a broad range of customers. The definition of a Large Scale Event is maintained by [TechOps](https://w.amazon.com/bin/view/TechOps), an organization that coordinates incident response across Amazon. When a Large Scale Event is ongoing, a flashing red strobe GIF (like a fire engine's light) appears in the top right-hand area of the TT site. Clicking the flasher takes you to [lse.amazon.com](https://lse.amazon.com), where you can identify the ticket associated with the event and join in tracking or assisting resolution.

We also classify services by the level of impact an outage of the service can have on the business. A service that can cause a SEV-1 impact is called a Tier-1 service. All other services are non-Tier-1, sometimes colloquially referred to as "Tier-2." We place higher expectations on Tier-1 services and service owners, both in meeting preparedness expectations, and in responding to SEV-1 incidents. When a SEV-1 incident is opened, Tier One Support (TOS) pages the on-calls of all Tier-1 services with the ticket information; they are required to check in to the ticket and join the associated conference call (listed on the ticket). Members of Tier-1 service-owning teams are expected to take Tier-1 Resolver training.

### More resources

* [Definitions of the impact classifications](https://w.amazon.com/index.php/GeneralImpactDefinitions) - Read more about the meaning and expectations for SEV-1 through SEV-5 here.
* [Tier-1 Service Definition and Notes](https://w.amazon.com/index.php/AvailabilityTier1Services)
* [Response expectations for different impacts](https://w.amazon.com/bin/view/TT/UserGuide/RemedyTicketEscalations/)
* [Service tier definitions](https://w.amazon.com/index.php/Tiers_of_Availability)
* Our traditional/legacy ticket tool, [Trouble Ticketing](https://tt.amazon.com) - note that this is going away at the end of 2021
* The newer [Simple Issue Manager (SIM) Ticketing](https://t.corp.amazon.com) tool -  all ticketing is transitioning to this in 2021
* [Tier-1 Resolver Training](https://w.amazon.com/bin/view/TechOps/Tier1_Resolver_Training/)

## The Consumer Availability Code

We have a set of important high-level guidelines for maintaining availability of Consumer services, called [The Consumer Availability Code](https://w.amazon.com/bin/view/Consumer_Availability_Code/). Think of it like a building code, or fire code: it changes over time, as we learn about new risks and best practices.

We strongly encourage you to read the code and to discuss any questions that arise about your own team practices with your manager and teammates.

## Consumer Reliability Engineering (CRE)

[Consumer Reliability Engineering (CRE)](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/) is an organization charged with improving the overall reliability of the software systems that power the Amazon Consumer customer experience.

CRE maintains resilience expectations for Consumer services and provides service owners with guidance, tools, testing, and inspections to improve the resilience of services.

CRE also runs the [Consumer Ops Review](https://w.amazon.com/index.php/Consumer/OpsMetrics) meeting for the Consumer organization. Modeled in part after the effective AWS Ops Meeting process, this meeting provides a forum for Tier-1 service owners and their leadership to collectively review Operational Excellence posture and to exchange operational insights and internal news relevant to service owners. All Consumer Tier-1 services are expected to be represented and prepared to speak to any [outages](https://outages.amazon.com/) in the previous two weeks (with a group [COE](https://w.amazon.com/bin/view/COE) review), and if randomly selected, be prepared to walk through their operational metrics deck. We also invite all other tech owners to attend as an excellent way to learn about successes, failures, and techniques that drive improvements in our availability and efficiency for customers.

We also encourage you to review CRE's [Best Practices](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Best_Practices/) and [Guidance](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance/) pages. These pages provide more specific guidance than the Consumer Availability Code for maintaining strong availability and resilience posture in your service. They also link to more material on availability and resilience topics, and we maintain and improve them more frequently as we learn.

## The Lite TRA self-assessment

The [Lite TRA](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/TechnicalRiskAssessment/TRALiteGuidance/) (Technical Risk Assessment) presents a self-assessment questionnaire for service owners. It provides a checklist to help gauge how well the service and team meet best practice recommendations. The survey asks questions about both software and operational practices of the team, points to guidance related to each question, and allows service owners to grade themselves against a "Green (good) - Yellow (warning) - Red (bad/strong warning)" rubric provided with the question.

Any service owner can complete the Lite TRA, but we expect ALL Consumer Tier-1 service owners to complete the Lite TRA for each service that they own and to keep it up-to-date. We enforce this in the lead-up preparation to each peak event.

In addition to the questionnaire, the Lite TRA process requires service owners to review the responses with an organizationally local senior engineer (either a Principal Engineer or Sr. SDE). The reviewer confirms that the team's responses are accurate, and helps to recommend and prioritize action items that will move any Yellow and Red items to Green.

CRE periodically updates the questions in the Lite TRA and works actively to automate inspections aimed at reducing the effort of completing and maintaining the Lite TRA information.

## Redundancy and isolation, Availability Zones, and VIPs

Redundancy and isolation of failures are among the most basic tools for improving availability.

Redundancy means having extra working capacity to handle failures. Isolation refers to any technique for reducing the impact of a failure to a smaller "blast radius". Isolation helps not only by limiting failure impact, but by giving us a basis for planning redundancy.

For example, AWS [Availability Zones](https://w.amazon.com/bin/view/Availability_Zone/) (AZs) isolate basic dependencies on infrastructure components (such as power, cooling, or network) to reduce the blast radius of failures of these fundamental dependencies. We can then plan for the loss of a single Availability Zone by allocating sufficient hardware across a set of Availability Zones to handle our full load, even with the loss of one AZ. Historically, Amazon expected our Tier-1 services to withstand the loss of any single Availability Zone. As of February 2020, this guidance has changed to only allow 90% of the redundancy necessary for handling an AZ failure, with the expectation that teams will have other levers to make up the difference. For complete details, see [Consumer AZ Redundancy Policy](https://w.amazon.com/bin/view/AZ_Redundancy_Policy/).

We regularly take advantage of redundancy by having many hosts behind load-balancing [VIPs](https://w.amazon.com/bin/view/VIP/). The VIP routes requests to the hosts set up "behind it," and does regular heartbeat "health checks" of these hosts. For a wide range of failures in one or a small number of hosts, the VIP can route requests to other hosts. Each VIP has a redundant backup load balancer, and Tier-1 services also use "partitioned" VIPs, both for redundancy and capacity. In Native AWS, [Application Load Balancers (ALBs) and Network Load Balancers (NLBs)](https://docs.aws.amazon.com/elasticloadbalancing/index.html) provide the load-balancing features that support this type of redundancy.

We also want to isolate problematic client traffic to protect well-behaved clients. Throttling requests on a per-client basis can help, but even throttling mechanisms can be overwhelmed.  Techniques like [shuffle sharding](https://aws.amazon.com/blogs/architecture/shuffle-sharding-massive-and-magical-fault-isolation/) work together with redundancy to limit the blast radius for a wide variety of traffic-driven and individual resource failures.

### More resources

* [Consumer's AZ Redundancy Policy](https://w.amazon.com/bin/view/AZ_Redundancy_Policy/)
* [Cell-based Architecture](https://w.amazon.com/bin/view/Cell-Based_Architecture)

## Deployment safety

We deploy changes to our Production environments with the intent of improving our services, but deployments of *any* changes to code or configuration also pose the points of highest risk of introducing a problem.

Gradual deployments help us isolate failures, reduce the blast radius of bad changes, and give us more opportunity to detect issues and roll back to our earlier working situation. Automated rollback, when tied to a service's key performance indicators (like availability and latency metrics), protects a service quickly without the need for human response. At the highest level, the Consumer Availability Code principles [Deploy gradually](https://w.amazon.com/bin/view/Consumer_Availability_Code/#HDeploygradually), [Roll back; Mitigate liberally](https://w.amazon.com/bin/view/Consumer_Availability_Code/#HRollback3BMitigateliberally), and [Only touch production with an MCM](https://w.amazon.com/bin/view/Consumer_Availability_Code/#HOnlytouchproductionwithanMCM) are the key guidelines to keep in mind when doing software deployments. Other specific recommendations are captured in our [Best Practices](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Best_Practices/#HDeploymentandOperations-relatedBestPractices).

We use [Dogma](https://w.amazon.com/bin/view/Dogma/) rules to identify specific risks and deviations from best practice in deployment pipelines, and in some cases to block pipelines that are not compliant with expectations.

Please see the *Deployment management* course under Operate with excellence in the [OE Learning Hub](https://w.amazon.com/bin/view/EE/Learn/OE/) to find more material about safely evolving and deploying your software. Among the lessons there, you'll find these gems:

* Keep changes small; break features into smaller increments. This reduces risk in each deployment, and also makes it easier to localize issues if they do occur.
* Avoid long-lived software branches. Project development branches should sync from the mainline frequently and merge back to the mainline as they produce those small increments you want to deploy.
* Pair new/updated functional tests with any changes.
* Build automated regression, stress, and chaos tests into your pipelines.
* Consider using toggles, flags, or dials to regulate the introduction of features. Remove them once the functionality has become the default.  This allows you a faster rollback mechanism than a software deployment, and can be used to limit the blast radius of bad changes.
* Be extra careful with database schema changes -- they can break rollbacks and require high-risk migrations.
* Deploy gradually with auto-rollback and no two regions at once! In each region, do an initial OneBox, then a gradual deployment.

### More resources

* [Pipelines](https://w.amazon.com/bin/view/Pipelines/)
* [Deployment management](https://w.amazon.com/bin/view/EE/Learn/OE/DeploymentManagement/)
* [Operational Excellence Learning Hub](https://w.amazon.com/bin/view/EE/Learn/OE/)
* [Consumer Deployment Safety Program](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/DeploymentSafety/)

## Stress testing and excess load shedding

VIPs provide our strongest line of defense against overload. By ["spilling over"](https://w.amazon.com/bin/view/Spillover/) traffic after reaching a maximum number of connections on which requests are active ("maxconns"), VIPs protect the backing host fleet from receiving more traffic when the fleet is completely occupied. This makes [setting maxconns properly](https://w.amazon.com/bin/view/Load_Balancers/MaxConns/) an important basic availability practice.

But what happens when you let too much traffic through to your service? A stress test is a test to understand how a service behaves under a load beyond the limits of normal operation (past failure). The service "breaking point" is the level of load at which performance degrades beyond acceptable levels (for example, an unacceptable error rate or unacceptable latency). A stress test should apply load beyond the breaking point, and observe and measure the service for expected handling over some period of sustained overload. We then measure the time the service takes to recover to acceptable performance levels when load is returned back to normal levels. A resilient service should remain alive, handling some fraction of its load while the overload is present, and shedding what is necessary for it to avoid collapse. When load is returned to normal levels, it should recover promptly and automatically, without any need to drop load well below normal levels.

Stress testing helps to find latent availability risks, like code that "blows up" when stressed and resource limits that we didn't anticipate hitting. Stress testing also confirms that the mechanisms we have put into place for load shedding really work. CRE's [Stress Test Toolkit](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/StressTestToolkit/) provides guidance on tools for stress testing request-response services.

Teams have also reported that comparing results from profiling (for example, with [Profiler](https://profiler.amazon.com)) with a normal load versus a heavy load has helped them to isolate where code is "blowing up" and helped to eliminate potential availability risks. One technique is to compare the two profiles, looking for regions that exhibit a disproportionately large increase (in time spent within that region) under heavy load, and then do a deep-dive analysis to understand the root cause of that increase.

You can learn more about [load shedding](https://w.amazon.com/bin/view/Services/LoadShedding) and [throttling](https://w.amazon.com/bin/view/Services/Throttling) from the [Building Reliable and Resilient Services Guide](https://w.amazon.com/bin/view/Services/).

### More resources

* [Load Balancers](https://w.amazon.com/bin/view/Load_Balancers/)
* [Guidance for Amazon-internal teams onboarding to Elastic Load Balancing (ALB/NLB)](https://w.amazon.com/bin/view/ELB/Onboarding/)
* [TPSGenerator](https://w.amazon.com/bin/view/TPSGenerator)
* [WindTunnel](https://w.amazon.com/bin/view/AutoPilot/Efficiency/Projects/AutomaticStressTest)

## Dealing with failure, chaos testing, and detecting cascading failure risk

There are roughly 460 Tier-1 services and many more non-Tier-1 services that combine to provide the Amazon retail customer experience. To maintain resilience across the site, we need to do more than just limiting individual service failures. We also need to prevent an outage or partial outage ("brownout") in any one service from cascading across the site.  **Individual service outages are bad; cascading outages are far worse.**

When a dependency fails, we expect a service to stay alive and to degrade as gracefully as possible, providing the best responses possible under the circumstances. When a dependency call provides some added value to the service's response, but is not essential for the service to provide a basic successful response, we say it is "non-critical." There are many cases of such dependencies in Amazon services. Your service should handle non-critical dependency failure, continuing to respond successfully and within your latency targets. When your service can't succeed and must send failure responses, it should fail as quickly as possible, avoiding throwaway work if the call will end up having to fail anyway. The service should stay responsive, and recover quickly when the dependency recovers.

Seeing how your service actually behaves under specific failure conditions provides the most trustworthy gauge of its resilience to those failures. Chaos testing (or fault injection) is our term for intentionally injecting failures to determine how a service deals with them. Because failure conditions tend to traverse rarely exercised code paths, chaos tests are a great way to find lurking issues that arise only under those conditions. The [CRE Chaos Engineering](https://w.amazon.com/bin/view/Chaos_Engineering/) team provides tools that allow service owners to run chaos tests on their own services.

As a basic target, service owners should aim to do at least the following two types of chaos tests:

1. Test the behavior of the service when the latency of responses from each underlying service dependency increases.
2. Test the behavior of the service when the availability of each underlying dependency drops.

This isn't saying service owners shouldn't test for more, but this is an important basic level to achieve because testing these types of failures can find these common cascading failure risks:

* **Excessively high timeouts in dependency calls.** If your service waits too long for a service dependency, and that dependency gets slow, your worker threads will all get occupied and be unable to handle your incoming traffic. Your service will start to fail. That's cascading failure.
* **Excessive retries on service failures.** You don't want your service to instigate a retry storm when your dependency fails. This can push a service that is "browning out" into a total outage. That's also cascading failure.
* **Incorrect treatment of non-critical dependencies.** You expected a dependency to be non-critical, but in fact it caused your service to fail or take too long to respond. That's another form of cascading failure.

Proper timeout, retry, and failure handling policies in your service are extremely important in preventing cascading failures. You should read the guidance on timeouts, retries, and failure handling in these references:

* [CRE Guidance](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance)
* [Timeouts, Retries and Jitter](https://w.amazon.com/bin/view/Services/TimeoutsAndRetries)

### More resources

* CRE Chaos Engineering's [Gremlin](https://w.amazon.com/bin/view/AvailabilityProgram/Gremlin/)

## Operational Excellence and effective event response

We build our services to be robust even in the face of failure, but we also have to be prepared for those cases where human response is required. In dealing with failures, we prefer resilience mechanisms built into services, then mitigation and resolution automation surrounding services, and finally human response, in that order. Well-designed automation works faster and with less error than humans, and the sooner we can detect and mitigate an issue, the shorter the period of customer impact.

Fast and effective human response relies on these essential ingredients:

1. Proper metrics, monitoring, and alarming to alert resolvers.  [Measure the right things](https://w.amazon.com/bin/view/CEH/Metrics/), then [put monitors with tight thresholds on them](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance#HMonitoringandAlarmThresholds), and alarm in a way that triggers the right resolvers to respond with the right urgency.
2. [Good dashboarding](https://w.amazon.com/index.php/Dashboarding) and diagnostic tools that help resolvers to understand the situation.
3. Prepared and tested runbooks/SOPs that guide resolvers in localizing issues and taking appropriate action.
4. Training and practice for resolvers.

When we set up our alarming, ticketing, dashboards, and runbooks, we want to tie these four ingredients together to minimize the time and effort of the resolver and to counter the "fog of war," the confusion in crisis that affects most of us humans (especially if we're paged in the middle of the night!). The autocut ticket for a given alarm should contain links to the appropriate monitors, dashboards, and runbook material that the resolver will need to respond to that alarm. The dashboard should have the right metrics to help the resolver answer basic questions. The runbook should suggest procedures referencing the dashboard to localize the issue and determine the appropriate mitigation, linking to an appropriate runbook to execute that mitigation.

The two key principles of high-severity issue resolution are:

* **Stop the bleeding!** Our priorities in high-severity response are always to mitigate customer impact first ("stop the bleeding"); root cause analysis comes later. Do just enough diagnosis to determine the proper mitigation action.

**Important:** Our default action in high-severity response is to roll back the most recent change, if it has a high likelihood of having caused the impact. Rolling back first mitigates the problem, and gives you time to figure out how to build and push out a more substantial fix.

* **Don't hesitate to escalate!** Escalation is your tool to call for help. There's nothing shameful in asking for help, especially when there is customer impact. Remember, as Amazonians, we're all trained to be customer-obsessed! This can mean getting help from your team by escalating to your manager, or getting help from many other teams by escalating to TOS.

### More resources

* CRE Guidance on [Monitoring and Alarm Thresholds](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance#HMonitoringandAlarmThresholds)
* [Oncall and event response](https://w.amazon.com/bin/view/EE/Learn/OE/OncallEventResponse/) online course
* Outage Analysis–[Appendix A: Outage Incident Taxonomy](https://w.amazon.com/bin/view/MedicalClass/OutageAnalysis/#Appendix_A:_Outage_Incident_Taxonomy)
* [Contacting TOS](https://w.amazon.com/index.php/TechOps#HContactingTOS)
* [Tier-1 Resolver Training](https://w.amazon.com/bin/view/TechOps/Tier1_Resolver_Training/)
* [Retail Call Leader Program](https://w.amazon.com/index.php/CallLeaders)
* Introduction to Operational Excellence: *Improve with metrics*: <https://w.amazon.com/bin/view/EE/Learn/OE/>
* Introduction to Operational Excellence: *Oncall & Event Response*: <https://w.amazon.com/bin/view/EE/Learn/OE/OncallEventResponse/>
* [Introduction to CloudWatch](https://w.amazon.com/bin/view/EE/Learn/AWS/Intro_to_CloudWatch/)
* [Monitoring and Logging with CloudWatch](https://w.amazon.com/bin/view/EE/Learn/AWS/MonitoringLogging/)
* [Monitoring and Responding to Events](https://w.amazon.com/bin/view/EE/Learn/SDE_Foundations/Monitoring) in SDE Foundations

## Consumer resilience score

CRE gathers data to maintain a [Resilience Score](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/ResilienceScore/) for each service. In a single number, the score reflects a very coarse measure of how well the service and service-owning team's operational practices meet CRE's resilience expectations. It incorporates sub-score components derived from Lite TRA assessments, stress and chaos test frequency and results, the absence or presence of identified risks in areas such as deployment safety, open COE action items, and other data gathered from automated inspections of the service's artifacts and resources. We require Consumer Tier-1 services to participate in this scoring; we encourage non-Tier-1 Consumer services to participate.

The [Resilience Score Dashboard](https://si.corp.amazon.com/dashboard/resscore) provides owners and leaders with a view of the resilience scores of services in their organizations, and the ability to drill down to understand the composition of those scores. This tool allows leaders to establish and drive goals to improve their service's availability posture and to identify areas that need more investment to improve their availability posture.

### More references

* [Resilience Score Dashboard User Guide](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/ResilienceScore/LeaderDashboard/)

## COEs and the COE review process

The Correction of Errors (COE) process seeks to improve the overall quality of our systems by documenting events to identify root causes and address them through trackable action items. It's our way of learning from our failures. When we have a customer-impacting incident or, in general, any incident or procedural "miss" that suggests an opportunity for improvement, we use the COE process to guide analysis and improvement.

The COE prompts us:

* To understand the incident and its customer impact, and to collect the metrics that can support further analysis
* To analyze how we detected it and what we could have done to detect it more quickly
* To analyze our response to the incident and how we could have mitigated or resolved it more quickly or more effectively
* To understand the timeline of the incident, and how it evolved as a way of noticing opportunities
* To perform a thorough analysis from each top-level deficiency exposed by the preceding analysis down to its root cause ("5 whys")
* To extract lessons we have learned and can teach others so that they can avoid similar issues
* To define corrective actions that will prevent similar incidents or improve our detection of and response to them

The Consumer organization has [a set of best practices and a deep-dive review process](https://w.amazon.com/index.php/Consumer/OpsMetrics/COE_Review) for COEs that are associated with outages or negative customer experiences of sufficient magnitude. The reviews bring together experienced Principal Engineers, the COE-owning team and their leadership, and other stakeholders to read, discuss, and refine the findings in the COE. This process is normally coordinated by Consumer Reliability Engineering, but teams can run [their own deep-dive reviews](https://w.amazon.com/index.php/Consumer/OpsMetrics/COE_Review#HAlternateDeepDiveReview) as well. After the review, COE owners present a summary in the Consumer Weekly Ops Review meeting to help spread the learnings.

## Cost efficiency

In eCommerce we operate in an ultra-low-margin business. Our ability to operate efficiently makes the difference between profit and loss, or between prices a customer is happy or displeased with. It is at the core of our business DNA, enshrined in both the Leadership Principle of Frugality and the [Amazon flywheel (lower cost structure)](https://improvement-ninjas.amazon.com/s3files/s3get.cgi/flywheel.png). The costs of running the consumer website are now measured in billions of dollars,  and it's our collective responsibility to ensure that we are realizing an economy of scale, rather than having scale kill our economy. This does not come automatically. If we don’t focus on efficiency, our infrastructure costs can grow faster than our business revenue.

Cost efficiency is about understanding how to run your use case as effectively as possible. At a minimum this means knowing how your service operates and making the right choices and tradeoffs that eliminate unnecessary waste. This applies to three areas:

* **Infrastructure:**  Are we using the right type and the right quantity at the right time and for the right duration?
* **Code:** Do we use the best algorithms, frameworks, libraries, and protocols for our needs? Are we using them correctly and in a frugal manner?
* **Architecture:** The whole is greater than the sum of its parts. How does our service fit into the big picture? Does our service do only differentiated (automated, reusable) work that matters?

## How is cost efficiency measured?

"Is my service efficient?" is a deceptively hard question to answer. We don't have a single, clear "one size fits all" efficiency metric, and we generally don't know how to interpret whatever metric we’re using. This forces us to talk in relatives rather than absolutes, such as  "is your service more efficient than last year?". Working in relatives rather than absolutes can easily create the wrong incentives. For example, a mandate to reduce <efficiency-metric> by 10% can be achieved trivially on an unoptimized service, but will be extremely hard to achieve on an already efficient one. Therefore, operating on a treadmill of continuous year-over-year targets should be avoided. Instead, absolute and consistent targets should be determined on dimensions that can be easily understood by everyone.

One thing common to all efficiency metrics is a correlation to costs measured in dollars, otherwise known as the [Infrastructure Market Rate (IMR)](https://w.amazon.com/index.php/IMR). If we can talk in terms of costs, with the denominator being “total” and the numerator being “useful,” we can set appropriate absolute targets for each of the three service areas mentioned in the previous section:

* **Infrastructure:** How much of the hardware are you using for processing? The most common bottleneck in hardware utilization is CPU cycles, although memory, network bandwidth, or disk bandwidth might also be the limiting factor. For example, if a service is using only 5% of the CPU and has 95% idle time, then the team is under-utilizing the hardware. This is measured and reported as the [Daily IMR Gap](https://w.amazon.com/bin/view/PerformanceEngineering/FleetUtilizationAnalyzer/Data/DailyIMRGap/) or DIG.
* **Code:** What percentage of compute resources like CPU, memory, disk, or network is your service using ? How much of these resources are spent running the actual business logic, versus remoting, tracing, metrics, and logging? Is your code implementation using the right libraries, data structures, algorithms, and caching techniques to optimize its demand for computer resources to do its job? The answers to these questions differ service to service, but tools like [Fleet Utilization Analyzer (FUA)](https://w.amazon.com/bin/view/PerformanceEngineering/FleetUtilizationAnalyzer/) and [Profiler](https://profiler-docs.aka.amazon.com/documentation/) can help you get more insights about them.
* **Architecture:** What's the ratio of unique and necessary work in a total call graph vs total work? Service-oriented architecture is great for opaque encapsulation, but can lead to significant waste. For example,  it might throw away already fetched data, only for another part of the graph to fetch it back again,  or pass around too much data that isn't needed.

It's also important to understand the interrelation of these targets. Only by optimizing infrastructure do you materialize reducing costs. Having efficient code minimizes utilization per unit of work; having efficient architecture minimizes total units of work by eliminating redundant work. Improving architectural or code efficiency will simply result in a corresponding decrease in efficiency of infrastructure. Therefore, to fully realize benefits from any code or architectural change, someone must also make an infrastructure adjustment. Note that this someone could be the infrastructure team itself; for example, [AWS Lambda](https://aws.amazon.com/lambda/) and [Honeycomb](https://learn.honeycomb.amazon.dev/) do it automatically as part of their offering. This isn't to say code and architectural improvements aren't worthwhile if you can't reduce infrastructure. Such improvements will still improve customer experience (latency) and improve the availability of your service. They just won't save Amazon any money.

We can also track our progress at a high level by looking at a business top-line output metric. For consumer, this is [$ IMR / Unit Served](https://w.amazon.com/bin/view/Consumer/OpsMetrics/Metrics/Efficiency/). The numerator here is the full IMR for the period, and the denominator is the number of units served (physical units shipped, digital products sold, and so on). At the top level of the business, it makes sense to set relative goals to improve (reduce) this metric year over year; however, it doesn't attribute easily down to organizations and granular services. So prioritizing investment areas to realize the improvement is done based on organizations being able to drill into absolute and consistent metrics, such as the aforementioned DIG.

## Scaling and efficiency

If our services only ever receive a constant load, it is relatively easy to cost-effectively size for the use case. Although constant and predictable load may apply for some use cases, most of the use cases experience varies due to both external factors (like customer requests) and internal ones (like clients onboarded to use your service). This variance may follow observable trends, such as a day/night cycle: traffic is lower in the middle of the night, as most users are asleep. Or the variance might be harder to predict--for example, during an unexpected event like the vast majority of users working at home due to COVID-19. Because efficiency is secondary to availability, the service needs to make sure we have sufficient capacity to meet peak demand, so the client doesn't experience the service being unavailable. However, being overscaled is a waste of resources; we have to minimize the chronic pain of being overscaled (inefficient) with the acute pain of being underscaled (unavailable).

Efficient scaling is a cross-cutting concern for our efficiency buckets of infrastructure, code, and architecture; we need to think about all three to be effective. This is why scaling details appear in each of the following sections, as the theme has different areas to inspect and different opportunities we can realize in each of these places.

# Efficiency and infrastructure

Your infrastructure is what your service runs on/the resources it consumes. This will change depending on:

* What type of service hosting you're using, such as EC2, Fargate/ECS, Lambda & API Gateway, Datapath, etc.
* What you're using for storage services, like local storage, EBS, S3, DynamoDB, Redshift, etc.
* The networking by which your infrastructure is reached, such as VPC, Private Link, ELB, etc.

Infrastructure often provides the most easily attainable efficiency opportunities, because it typically requires only a small configuration change in a UI or API call to reduce the number of hosts, reduce retention windows, or change instance types. To examine your infrastructure efficiency, first review your bill--are you paying for things you don't need? Next, review your utilization--are you paying for things you're not utilizing? Finally, understanding how you scale up/down as the load changes helps you identify inefficiencies. For example, you might preemptively overscale, or not downscale in response to reduced volume and/or client load.

## Your bill

The bill often tells you a lot about your application, and can contain surprises that lead to quick wins. The bill is aggregated by fleet hierarchy in the [monthly-statement portal](https://monthly-statement.amazon.com/). As you work through your bill, it's worth asking yourself some key questions:

* **Are the big hitters on your bill what you’d expect? Are they in line with financial drivers?** If not, you may well have a "leak." For example, retaining data indefinitely in storage services such as S3 or DynamoDB might result in your bill increasing over time, even if your business drivers remain the same.
* **Are there any large line items on the bill that you don’t know or weren’t expecting?** Some services can cause unexpected “knock-on effects.” For example, the AWS Key Management Service (KMS) emits a cloud trail event that gets processed by AWS GuardDuty. Prime Video found they had a $25K monthly bill for KMS and a further $8K for GuardDuty, because they had not enabled KMS key caching. [Enabling key caching for 5 minutes saved them $32K/month!](https://quip-amazon.com/vf0ZAyVaDw6w/Cost-savings-from-caching-KMS-keys)
* **For the bigger items, look at the detailed bill.** Getting into the attribution details is often illuminating. For example, Dynamo DB read/write capacity, storage billing, streams, and net transfer might provide a clue for optimizations that can be made. To get more detailed billing information, you'll need to check in the AWS account(s) used for the fleet. First, find the account through [access.amazon.com/aws/accounts](https://access.amazon.com/aws/accounts) and click **Console Access**. To work with billing inside the AWS console, see the AWS Blog post about [getting started with billing](https://aws.amazon.com/blogs/aws-cost-management/back-to-basics-getting-started-with-the-billing-console/). For Software as a Service offerings, such as SQS, S3, and so on, tools like [AWS Trusted Advisor](https://aws.amazon.com/premiumsupport/technology/trusted-advisor/) can surface [many common inefficiencies](https://aws.amazon.com/premiumsupport/technology/trusted-advisor/best-practice-checklist/), such as API calls where [SQS is not using long polling](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/reducing-costs.html).

## Utilization

What does your service look like against primary resources, like CPU, Memory, and IO (both Disk and Network)? For storage services, including for your telemetry, you should also know how much of the planned capacity you're using in services like Dynamo DB or RDS instances, and how much storage space you use. You can get this information by checking metrics you or your service provider emit in CloudWatch or [MonitorPortal (MAWS)](https://w.amazon.com/bin/view/MonitorPortal/), or by using an aggregation tool such as [Fleet Utilization Analyzer (FUA)](https://w.amazon.com/bin/view/PerformanceEngineering/FleetUtilizationAnalyzer/). Ask yourself these key questions:

* **What is your application constrained on--CPU, Memory, IO, or “Other” ?** The answer is often “Other,” which generally means the application is at least bi-modal. In other words, it  constrains differently under exceptional circumstances,  rather than how it behaves in normal operation.
* **How are you determining constraint?** For example, if “median of max CPU” (see [Daily IMR Gap](https://w.amazon.com/bin/view/PerformanceEngineering/FleetUtilizationAnalyzer/Data/DailyIMRGap)) is only 30%, and it's regarded as the constraining factor, why can't you descale the service? As a rule, we should aim for [55%](https://w.amazon.com/bin/view/Consumer_Tech_Finance/BRP/Guides/101/) for online request/response services that ultimately support a user or external API, and as close to 100% for offline/asynchronous services as can be pragmatically achieved.
* **Are you using the right form factor or instance type?** If you're memory constrained, but running a compute-based instance such as C5, why not move to a memory-optimized instance? Tools like [AWS Mettle](https://w.amazon.com/bin/view/AWS/Mettle/) can help analyze your application’s utilization and recommend appropriate instance and lower-level container types. This is also true for storage services; -"hot," key addressable stores like DynamoDB are much more expensive than colder stores such as S3 and Glacier.
* **How long do you need to retain data?** You may have [business requirements mandating how long you must retain data](https://inside.amazon.com/en/services/legal/us/documentretention/Pages/DocumentRetentionPolicy.aspx), but often you'll retain data "just in case" it's needed. The longer you retain data, the more you pay for storage cost; however, the older the data, the less likely it is to be accessed or useful. Each service has different storage needs, so make sure you have metrics on the age of records being accessed (such as when it was last updated) and set your retention window to an appropriate trade off between the length (cost) of the storage window and the percentage of access requests you want to be able to support.

## Scaling

Load for a service fluctuates and clients come and go. How quickly and stably your service scales correlates directly to how efficiently you can operate your service. In an ideal world, everything would be on-demand per request, so you never pay for overhead or idle time. This doesn’t work for many services today, but the more automated your scaling is, the more the waste area under the curve can be reduced and so forms an important efficiency consideration. To learn more about achieving efficiency through scaling, see [Horizontal scaling](https://w.amazon.com/bin/view/CEH/Scalability/#HHorizontalscaling) in the Scalability section of this handbook.

# Efficiency and code

Our application code can be thought of in three major buckets:

* The unique business logic we're looking to run as our value proposition
* Remoting/Framework code that supports getting work and remote dependency calls to/from our business logic
* Telemetry code that allows us to monitor and operate the service, "in sickness and in health"

The primary tool for looking at application code is a profiling tool. The [AWS CodeGuru Profiler](https://docs.aws.amazon.com/codeguru/latest/profiler-ug/what-is-codeguru-profiler.html) tool has a Points of Interest (POI) feature that can automatically find and recommend solutions or alternatives for many common efficiency problems. For more information, see the [(2020/3) AWS Blog post on Profiler](https://aws.amazon.com/blogs/machine-learning/investigating-performance-issues-with-amazon-codeguru-profiler/) or the [(2016/7) POA talk](https://broadcast.amazon.com/videos/46242).

## Business logic

The business logic is the reason the service exists. Its logic is proprietary to this service and only you, as the service owner and domain expert in it, can determine whether what it's doing is correct and as expected. Despite each service's logic being completely unique code, there are still several common considerations that are worth looking out for:

* **Is time being spent where you expect it?** As you look at where the time is spent within the business logic, does it align to where you expect the computational complexity to exist? If bits you expected to be fast are taking a disproportionately large period of time, they may be doing more than you expected, and more than you potentially need. It’s worth inspecting them to see if a more lightweight approach can suffice.
* **Are you using the best library for the job?** While your business logic itself is proprietary and unique, there is a high probability you'll be using a number of common libraries for core functions and algorithms. These can often be sub-optimally configured, or have alternatives with different capabilities or speed tradeoffs, such that correctly configuring or replacing them can yield significant efficiency benefits.
* **Can you safely introduce state?** If your code is doing repetitive work within or between units of work, is it safe to retain the outcome of the work to re-use? This could be as simple as memorizing or having a singleton for a compute resource, having a materialized configuration by introducing caching of calls, or even having a durable stateful coordination. For example, the Jackson Object Mapper is Thread Safe after configuration. However, it's more common to [instantiate a new Object Mapper every invocation](https://profiler.amazon.com/efficiency-pattern?patternId=43967f61-9715-4fd3-a22c-c647559b3613) rather than having a singleton instance. Storing an instance as a singleton allows Jackson-type caches to work and can considerably reduce processing time.

## Remoting

Invoking your business function running inside a service generally requires first receiving and then transmitting. Likewise, for each remote dependency you need to invoke, you'll be doing the same, just the other way around--transmitting and then receiving:

* **Receiving** involves:
  1. receiving bytes over a network,
  2. decompressing,
  3. decrypting,
  4. authenticating,
  5. deserializing, and finally
  6. invoking your business function.
* **Transmitting** does the symmetric inverse operations in the opposite order:
  1. invoking the client call,
  2. serializing,
  3. signing,
  4. encrypting,
  5. compressing, and
  6. transmitting.

These components will often be assembled in a fairly narrow set of commonly used frameworks, such as Coral or ARest for Java applications in Amazon, and in turn have a relatively converged set of libraries and protocols operating each one.

As our services become progressively more "micro," the amount of work spent remoting will often increase relative to the time spent doing the explicit business logic. As such, remoting will often be an area where significant efficiency opportunities can be realized by ensuring that you’re using the latest versions of the right tools correctly. Think about these points for your remoting components:

* **For receiving/transmitting:** What kind of I/O model are you using, synchronous or asynchronous? How many concurrent requests are you able to handle? What happens when you exceed the amount? We're looking for situations where resources will be exhausted and the application will start queuing somewhere. This may be a receiver threadpool, or a queue.
* **For decompression/compression:** How large are your uncompressed payloads? If they're relatively small (say < 1KB), compression may be unnecessary or even of negative value. What library or libraries are you using for compression? Many hardware/OS/VM configurations have native instruction support; if it's available, make sure you're using it as it can make a big difference.
* **For decrypting/encrypting:** Are you using payload encryption, such as AAA Symmetric, or channel encryption, such as TLS? This may affect whether you're doing decryption/encryption ahead of process, as with CDORelay, or in-process. Much as with decompression/compression, using a facility that can take advantage of native hardware/OS support, such as the [Amazon Corretto Crypto Provider (ACCP)](https://w.amazon.com/bin/view/AWSCryptography/AWSCryptoTools/ACCP/), rather than software emulation can make a big difference. Likewise, if you see large volumes of SSL Handshakes being established, it's likely worth looking at how connections are established and how long you keep them alive and allow them to be reused.

**Note:** Longer life may provide better performance, but could degrade availability by reducing the ability to load balance.

* **For authentication/signing:** Authentication/signing is necessary to protect your clients; however, if you're spending a lot of time on authentication/signing, it might be due to excessive remote calls caused by ineffective local caching, or sub-optimal configuration of your framework. For example, are you authenticating for both CloudAuth AND AAA in Coral, even if the CloudAuth check runs first and succeeds?
* **For deserializing/serializing:** Your choice of serialization protocol, your library/version, and ensuring that you’re using the library optimally can be critical. For instance, Java Default Serialization is [a protocol that is verbose and slow](https://dzone.com/articles/avoiding-java-serialization), yet still widely used. Consider migrating to a different protocol if it's taking a significant amount of server time. Likewise, CoralRPC's [legacy processing implementation is also very slow](https://w.amazon.com/bin/view/Tim_Griffith/Profiling/CoralRPC/). It was rebuilt with [Ditto](https://w.amazon.com/bin/view/DC2.0/Ditto/) to use [Ion](https://w.amazon.com/bin/view/Ion/) under the covers, which greatly improved performance, but there are still [many users running the legacy CoralRPC implementation](https://profiler.amazon.com/efficiency-pattern?patternId=380de249-7a9e-4f9e-8a78-0236716b4d15). Ion is a good example of a protocol that is often used sub-optimally. Ion supports two primary modes, [Text and Binary](http://amzn.github.io/ion-docs/docs/spec.html), where binary is considerably more efficient than text, especially to transmit and deserialize. In addition, Ion Binary supports two means of providing the encoding symbol table: locally, passed by value in the payload, or shared, passed by reference that the recipient is required to resolve. Services regularly operate Ion text rather than Ion Binary, not understanding the difference between local and shared symbol tables, and so incorrectly fearing that their clients won't be able to decode. Switching over to a binary encoder is a 1-line code change that requires no corresponding consumer or client changes and can yield significant efficiency opportunities. In one extreme case this saved hundreds of thousands of dollars yearly.
* **Invoking the business function:** The invocation of the business function itself can be significantly expensive. For example, Coral looks to create an instance of the activity class on each request/invocation from the Dependency Injection framework, such as Spring or Guice. This is lightweight when all the rest of the components that will be wired are pre-instantiated singletons. However, when you’re applying [cross-cutting concerns](https://w.amazon.com/bin/view/JSR-269/Cross-cutting_Concerns/) like metrics, caches, retry interceptors, etc. by annotation, or not realizing that some DI frameworks such as Guice or Dagger don't create singletons unless explicitly instructed to do so, the invocation processing can be extremely expensive. In such cases, investigating and resolving often yields a quick and meaningful efficiency win.

## Telemetry

Telemetry is the data we emit from our service to be able to effectively operate the service. We emit metrics for monitoring, both for manual inspection and to create alarms that can cut tickets or page us in to look at the service when it's behaving abnormally. We emit logs to provide more detail as to the data flowing through our systems and the decisions we're making. We use these logs infrequently for diagnosis, or may conduct automated behaviors over them to create and emit metrics. Finally, we use tracing to understand interaction and data flow over the wider ecosystem in which our service participates. This allows us to understand and diagnose behavior occurring across our service-orientated architecture holistically. Telemetry is critical for effective operations, but it can be expensive, and so it's fairly easy to end up being unnecessarily inefficient with your telemetry by generating excessive volumes or unnecessary or redundant telemetry.

There are several key themes across all telemetry:

* **Sampling:** You need to sample enough to get a decent representation of all traffic; however, producing for every request is often unnecessary and so increases costs without providing any greater utility to the operator.
* **Reporting:** Ensuring reporting is done efficiently and asynchronously to the business logic can make a significant difference to both your service's availability and its efficiency.
* **Use it sparingly:** There is such a thing as "too much instrumentation." Telemetry that doesn't get used is worse than useless; it increases costs and, in the extreme, can even degrade availability. Output what you need, nothing less and nothing more. Ideally, dimensions you emit should follow common conventions so that your clients can understand your service, and you can understand your clients and your dependencies.
* **Retention:** Holding onto historical telemetry costs money, but you should balance that cost against legal and operations requirements. Work out how long you *need* to retain information for, but don't keep logs for any longer. By example, [Timber recommends](https://w.amazon.com/bin/view/Timber/LogRetentionSettingRecommendations/#HRecommendations) 10 years for audit logs, 3 months - 1 year for production debug logs, and < 3 months for non-prod debug use cases.

### Tracing

Tracing, such as that provided by [AWS X-Ray](https://aws.amazon.com/xray/) and [Axon](https://w.amazon.com/bin/view/Axon/public/Overview/) (formerly GAT), is extremely powerful, but also the most expensive form of telemetry. It allows you to see requests not just from view of your service, but across the entire application architecture, so that you can understand request patterns and their latency distribution end to end. This end-to-end nature also means your service forms a link in the chain: if it doesn't participate, the trace will be incomplete, which reduces its utility for upstream and downstream systems. Likewise, you have to consider how much load your decision to trace can generate for downstream systems. If you call a dependency 10x for each request you get and request a trace on each call, you're signing your downstream up to 10x the tracing burden you are paying... or else risking them breaking the chain and resulting in partial or incomplete traces. The sampling considerations for tracing are more complex and critical than those for any other type of telemetry, because you have to think holistically, not just locally. Key considerations for tracing include:

* **Sampling:** How have you configured trace [sampling](https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-sampling)? If your application fans out calls, and your tracing tool supports it, are you using sub-sampling to keep from overloading your downstream dependencies with requests and increasing the size and cost of each trace? Depending on the load of your service, you might only need 1 in 10 (or 100, or 1000) requests to get a representative distribution of client traffic.

### Metrics

Metrics, such as those provided by [CloudWatch Monitoring](https://aws.amazon.com/cloudwatch/features/), are used to emit counters and timers for functions that you wish to monitor over [dimensions](https://w.amazon.com/bin/view/Coral/Metrics/Manual/MetricsLine/#Dimensions) that are useful for more granular understanding. These metrics are recorded as a time series that can be used to [build operational dashboards](https://w.amazon.com/bin/view/Monitoring/Documentation/DashboardingBestPractices) and [monitors](https://w.amazon.com/bin/view/Monitoring/Documentation/Carnaval) that can programmatically raise [alarms](https://w.amazon.com/bin/view/MonitoringTeam/Alarms) when metrics exceed a threshold for a sufficiently long period.

While metrics are specific to a given service, there are a number of conventions that either must or should be followed. When using MetricAgent, you must provide [4 mandatory metrics (StartTime, Time, Program, and Marketplace)](https://w.amazon.com/bin/view/MonitoringTeam/Alarms), or the metrics will be ignored and not shipped to CloudWatch. When writing capturing-dependent service calls, you should use the enumerated [Service Metrics](https://w.amazon.com/index.php/DSE/BSF:Monitoring:Service_Metrics), as these allow for [broadly understandable and consistent reasoning about your service](https://share.amazon.com/sites/ASCS-SCP/_layouts/PowerPoint.aspx?PowerPointView=ReadingView&PresentationId=/sites/ASCS-SCP/Shared%20Documents/Platform/Development_Practices/Service-Metrics.pptx&Source=https%3A%2F%2Fportal2010%2Eamazon%2Ecom%2Fsites%2FASCS%2DSCP%2FShared%2520Documents%2FForms%2FAllItems%2Easpx%3FRootFolder%3D%252Fsites%252FASCS%252DSCP%252FShared%2520Documents%252FPlatform%252FDevelopment%255FPractices&DefaultItemOpen=1&DefaultItemOpen=1).

Key considerations for efficiency metrics telemetry, in descending order of average efficiency opportunity, include:

1. **Inventory:** How many different metrics is your service producing? Metrics are expensive, and producing metrics that aren't part of active operations flow is a waste of local service time and can unnecessarily bloat the CloudWatch bill. MAWS services that haven't yet been migrated to CloudWatch aren't free, they just aren't being billed to the service owner. Amazon is still paying the cost overall. Reviewing the metrics that are being produced and removing any that are no longer used or necessary, such as those produced for a specific feature investigation that has already shipped, can save considerable local compute and monitoring service bills.
2. **Scope/batching:** Metrics are much more effectively transmitted in small batches, to either the CloudWatch API or the [Service Query Log](https://w.amazon.com/bin/view/Monitoring/Documentation/ServiceQueryLog) intermediate form. Flushing too frequently can cause significant bottlenecking in the service codebase, especially if the submission mode is synchronous (for more information, see submission mode later in this list). A common efficiency problem has CoralMetrics flush for each internal function call, rather than aggregating into the request metrics and flushing to disk once per service request. A key litmus test is to look at the total number of submissions, be it API submissions counted through metering or entries in a service log on disk, relative to the number of business value units. If you have a ratio much higher than 1, there is likely an efficiency opportunity available.
3. **Sampling:** Sampling for metrics can greatly reduce the cost of operations, but can make reasoning about the metrics more difficult. Making a consistent decision across all metrics at a consistent business value grain, such as “per service request,” can help keep reasoning about the service simple and consistent. Facilities exist for major frameworks such as the [SamplingReporterFactory in CoralMetrics](https://w.amazon.com/bin/view/Coral/Metrics/Manual/#Sampling).
4. **Capture:** How do you emit the metrics? Do you flush to disk and ship with the [MetricAgent](https://w.amazon.com/bin/view/MonitoringTeam/MetricAgent/), or do you publish directly to the CloudWatch API? The former can lead to contention between metrics and logging for the disk, and can result in making a process that otherwise would be compute-bound become I/O-bound and able to serve a far lower TPS, significantly increasing request latency.
5. **Submission mode:** Do you submit metrics on the processing thread synchronously, or hand off to an asynchronous thread to submit? Doing the latter is often great for efficiency, but comes with some key considerations, including how big a handoff buffer is allowed and what happens when it’s filled. For example, does it block back into the processing thread, or drop?

For detailed information about these and other metrics to track, see [Metrics and measures](https://w.amazon.com/bin/view/CEH/Availability/Metrics/) in this handbook.

### Logging

Logging captures detail about work that is executing within your process. It can be used as the raw form over which metrics are processed and extracted, such as for Lambdas or with LogScanner and MetricsAgent. However, it’s most commonly used for developer operations, to analyze what's happening after metrics show something abnormal or unexpected is happening. Normal operations make logging, even more than metrics or tracing, a write once, read (almost) never endeavor. However, when logging is read - it's often in a time-sensitive or business critical situation. Key logging considerations include:

* **Log level (logging too much):** How much are you logging each hour and day? Are you logging at too fine a level for more of your code than you need? Running [granular debug or trace levels](https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm) in prod generally lowers efficiency, and can present significant latency and availability risks as well, including filling up the local disk, which breaks most forms of server processing until the disk is cleared out. Setting log levels appropriately can be hard, and it’s common to under-log by default. If you then have to reconfigure and restart your process in the event of failure, you spend valuable time, and might also clear the root cause for the malformed behavior and make diagnostics even more difficult. We recommend scoped logging, sampled per business value unit/request. For example, you can do this in Log4j2 by setting a sample flag in its [ThreadContext](https://logging.apache.org/log4j/2.x/manual/thread-context.html) and configuring a [ThreadContextMapFilter](https://logging.apache.org/log4j/2.x/manual/filters.html) to enable more granular levels when sample is set to **true**. This lets you exercise the more granular logging consistently in production across the lifetime of a BVU/request, without overloading your service.
* **Encoding:** Do you use structured logging? Unstructured logging is expensive to generate due to formatting costs, verbose (because it’s formatted for human reading), and so large that it’s costlier to compress, ship, process, and store. It's also slower and more expensive to query/read. If you're using unstructured logging, consider migrating to structured logging. [Shorthand](https://w.amazon.com/bin/view/Shorthand/ApplicationLogs/) is an Ion-based structured logging format that is being rolled out across key logging services, such as AWS Scroll.
* **Throw-away work:** Are you generating log messages prematurely, or applying string formatting too soon? If you're formatting a complex message into a string before passing it to the logger, that work is wasted. The logger immediately throws it away if it sees the log level isn't enabled. To avoid this, always use [parameterized logging](https://logging.apache.org/log4j/2.x/performance.html) facilities for your runtime. This ensures that the log message isn't formatted or encoded until it's past the logger level check.
* **Retention:** What's your retention policy, and why is it as long as it is? Retention periods may be set much higher than practically needed. This is particularly true for resources such as the internal [Timber](https://w.amazon.com/bin/view/Timber) log archival service, which doesn't (currently) charge back its customers and so isn't forcing appropriate cost/benefit tradeoff decisions for service owner teams.

### Profiling

Profiling helps you understand how your service is operating holistically. Profiling can be done for:

* **CPU:** what’s being executed
* **Memory:** what's being allocated and retained
* **I/O:** what's being transmitted and received

In all cases there are two primary types of profiling: sampling and instrumented. Sampling involves periodically polling a resource to build up a statistical picture of the application overall. For example, you might poll a resource to get the stack of each registered or running thread, or to take a snapshot of a segment of memory. Instrumentation involves modifying the code itself to explicitly capture information, such as times per call or exact allocation sizes.

Instrumentation is higher fidelity and can give you information that sampling cannot. An instrumented CPU profile can give you counts, so you can distinguish between a method that is called very frequently and a method that itself takes a long time to execute. Instrumentation can also be used for latency profiling, which a sampling CPU profiler cannot because it’s entirely agnostic of any given request or transaction. Instrumenting generally has very significant overheads and so materially increases the cost of executing the code itself and will reduce the performance of the running code. As such, instrumentation is preferred for local investigation, or used very sparingly and in a highly targeted fashion in production. Capturing timing metrics or initiating tracing spans for critical functions, such as a service method or outbound request, are essentially human-implemented, highly selective instrumentation that runs in production.

Amazon provides a low-overhead, always-on sampling CPU profiler that is suitable for production use cases: [AWS CodeGuru Profiler](https://docs.aws.amazon.com/codeguru/latest/profiler-ug/what-is-codeguru-profiler.html) and its internal counterpart [Amazon Profiler](https://profiler-docs.aka.amazon.com/documentation/). We strongly recommend enabling it for your services to support production operations and application efficiency improvements. If the profiling itself is having a noticeable overhead, such as > 1% of CPU, you should walk through the [Profiler troubleshooting guide](https://profiler-docs.aka.amazon.com/documentation/getting-started/troubleshooting.html).

## Scaling

How you write your code can have a big impact on your ability to scale. The [Scalability](https://w.amazon.com/bin/view/CEH/Scalability/) section of the CEH goes into detail about two important criteria:

1. [**How fast do you start up?**](https://w.amazon.com/bin/view/CEH/Scalability/#HStartuptime) The faster you can start up, the more you can scale reactively, with only the exact amount of capacity you need. It's better than trying to guess, or to keep extra safety capacity in reserve in case you underestimate!
2. [**How much load can one instance take?**](https://w.amazon.com/bin/view/CEH/Scalability/#HBreakingpoint) The more load an instance of a service can take before it starts to impact performance, the lower the scaling gradient can be for your service, making it easier to keep up with changes in demand.

# Efficiency and architecture

Service-oriented architecture and two-pizza teams allow us to build quickly and have strong ownership, but such decentralized autonomy can lead to significant inefficiencies when looked at holistically. These include load shaping, unnecessary repetition, unused computation, and redundant capability. Solutions for these inefficiencies all have in common the need to better understand the wider architecture and domain that the service operates in, even when the service appears to be optimal locally.

## Load shaping

Not all services have the same level of criticality, and not all services have the same responsiveness requirements. For services that are less critical or that don't have an immediate end user, or offline processes like routine ML model retraining or end-of-day job processing, you can adjust your schedule. Scheduling processes at [off-peak times](https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/) and jittering scheduling can significantly [improve efficiency by demand-smoothing](https://w.amazon.com/bin/view/DataForge/Motivation/#Minimize_Hardware_Waste_.28Company-Wide.29) across services. This section currently focuses on offline processes, but the idea of giving up capacity in emergency situations from less critical fleets applies to all service types. Key considerations for planning your service’s schedule include:

* **How and when do you schedule your offline processing?** Are you registered in a system that understands your service’s criticality and delivery needs? Are you able to run your jobs asymmetrically during the day, or in different availability zones (AZs) to take advantage of regional capacity? This, of course, assumes your input/output datasets aren't massive and won’t incur significant bandwidth costs.
* **Is your offline processing appropriately jittered as to when it starts?** Humans like converging on human-understandable/communicable concepts. This means there is a propensity to schedule events on time boundaries, such as [on the hour or at midnight](https://paste.amazon.com/show/tpgriff/1580762180). This creates correlated spikes in demand (if using on-demand capacity) and power consumption. Idle capacity doesn't consume much power, while intense jobs running at the same time do.
* **Can you batch work?** Batching up multiple requests can share fixed overheads such as instance creation time, network establishment time, header overheads, and more. But bigger batches aren't always better, as they trade off sub-linear efficiency gains against linear latency, and very large batches can reduce process stability unless you design for them explicitly.

## Unnecessary repetition

The services we create, be they request/response or [Elastic MapReduce (EMR)](https://aws.amazon.com/emr/) jobs, are generally strongly owned and encapsulated. This is great because it allows for parallel evolution and decouples clients from the details of how we implement and what dependencies we're using. However, it often leads to some repetitive patterns like:

* Under-utilizing caching data that will be re-fetched multiple times to serve the same request, just via multiple different service paths.
* Context passing, where the dataset gets more and more columns or entities added to it as it passes through dependencies, just in case the consumer needs them.
* Further retries. These not only risk availability, they can often be unnecessary, caused by lack of client understanding.
* Replication. The number of copies and the form factor we hold them in depends a lot on the distribution of data criticality.

### Caching

Caching is based on the concept of not replicating work that was recently performed. By placing a cache in front of a service, we may be able to serve some responses from the cache and avoid re-computing the data. The way this works is when a caller requests a resource by some key, the service first consults the cache. If the key matches an entry in the cache, the result is returned without re-computation. If no entry exists in the cache, the service computes the result, returns it to the caller, and inserts an entry in the cache so that it is available for future callers. Caching is most applicable for frequently read data that changes relatively infrequently. While caching primarily has latency benefits, it also plays an important role in scaling by reducing load to services. As the following diagram depicts, caches may exist at multiple layers of the infrastructure depending on the use case.

Diagram of the layers of the infrastructure at which caching may be used, such as on user desktop and mobile devices, in networking devices on the Internet such as DNS and CDNs, in application software, and in databases.

| **Layer** | **Client-side** | **DNS** | **Web** | **App** | **Database** |
| --- | --- | --- | --- | --- | --- |
| **Use case** | Accelerate retrieval of web content from websites (browser or device) | Domain to IP Resolution | Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server side). | Accelerate application performance and data access | Reduce latency associated with database query requests |
| **Technologies** | HTTP Cache Headers, browsers | DNS Servers | HTTP Cache Headers, CDNs, Reverse Proxies, Web Accelerators, Key/Value Stores | Key/Value data stores, local caches | Database buffers, Key/Value data stores |
| **Solutions** | Browser-specific | Amazon Route 53 | Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions | Application Frameworks, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions | ElastiCache for Redis, ElastiCache for Memcached, Guava for local, in-memory caches |

Caching solutions require some careful consideration. Effective caches can provide better latency and efficiency, but do so by trading risks on *freshness* of data and introducing *bi-modality* into your service.

#### Cache effectiveness

Cache effectiveness comes from the probability of finding a relevant item already in the cache and thus avoiding the work necessary to reproduce or reload the item. This probability is referred to as the Cache Hit Rate and is a function of:

* **Size:** The bigger the cache, the more items it can hold, and so the greater probability that a request will include an existing cache item.
* **Replacement policy:** When the cache is asked to store more items than it can accommodate, which value(s) should it evict? Common strategies are Least Frequently Used (LFU) and Least Recently Used (LRU), but there are [many strategies to choose from](https://en.wikipedia.org/wiki/Cache_replacement_policies) depending on your use case.
* **Retention policy:** How long should a key be retained for before it expires and a new version should be loaded/calculated? While some caches will be indefinite, most caches will use an associated time-to-live (TTL), after which the cache record is said to be expired.

Teams should use load test results and production performance metrics (CPU, cache hit rate, and so on) to tune cache size, replacement policy, and retention policy to strike the best balance of efficiency and freshness. Teams must capture cache hit metrics as part of their [overall service metrics](https://w.amazon.com/bin/view/CEH/Metrics/), and have associated thresholds and alarms to alert when depressed cache hit rates pose an availability risk.

#### Data freshness

Results returned from a cache run the risk of not being up to date. If a requested resource has changed since the time it was added to the cache, the caller will receive “stale” data. The longer the TTL or the higher the update frequency for the key, the more chance there is of serving stale data. To mitigate this scenario, TTL must be set appropriately and requires coordination with the data owner to understand the business implications of stale data on the customer experience. Some data, such as marketplaces, rarely, if ever, change and can be cached indefinitely. Data such as product details may change infrequently and can have longer TTLs (hours or days). Other data, such as delivery options, which depend on more frequently changing factors like inventory levels, must have shorter TTLs (seconds or minutes). Note that where services backed by a cache are under duress, serving stale data is often preferable to returning errors--but depends on the implications of stale data to the use case.

Additionally, in certain circumstances, service owners must intentionally remove cached data to protect the integrity of the customer experience. Consider a scenario where some incorrect product data has been cached, and the TTL is some time in the future. For example, in a recent COE, a bug introduced data pollution into a cache, and the cache then perpetuated the problem for the duration of the TTL. In these cases, it may be best to explicitly remove cached entries instead of waiting for them to expire.

#### Bi-modal operation

Introducing a cache gives your service two operational modes: 1) responses that are served from the cache and 2) those which go on to do full processing. These operational modes will often have significantly different behavior and costs (latency, CPU, and so on). Service owners must ensure that their services do not rely exclusively on caching for either scaling or latency. Consider the following scenarios:

* **Cache becomes unavailable.** The risk of service overload increases as traffic that is normally absorbed by the cache flows through to the full service processing. Even if the service is able to absorb the additional load, responses will be delayed due to the additional work being performed. This may result in caller request timeouts, which may result in request retries. In this scenario, if the initial requests don’t overload the service, the retry storm likely will.
* **Abnormal request traffic.** If the service receives an outsized number of requests, the cache may be forced to start replacing keys before they are next requested, leading to cache misses and an increase in requests being fully processed.
* **On startup/while "priming".** Services that rely on caching local to the process need to throttle the rate at which they accept traffic while the caches “warm” during startup, because they won't have the same cache hit rate as already-warmed servers and so will perform poorly relative to the running fleet.

**Important:** Caching does not obviate the need for [load-shedding mechanisms](https://w.amazon.com/bin/view/CEH/Availability/#HStresstestingandexcessloadshedding) such as request throttling and circuit breakers. On the contrary, caching usually means you absolutely need load shedding, because when your caches fail, so will your service, if you were relying on them *without* any form of load shedding.

The only example where this isn't true is when you're using caching only for latency reasons, and you let every request, even if it's a cache hit, end up with a downstream call to the data source.

#### More resources

Many caching technologies exist at Amazon with integrations for commonly used frameworks. There are very few, if any, good reasons to re-invent the caching wheel. Instead, take a look at these resources:

* [Caching Overview](https://aws.amazon.com/caching/)
* [Caches](https://w.amazon.com/bin/view/Caching/#Caches)
* [Caching challenges and strategies](https://aws.amazon.com/builders-library/caching-challenges-and-strategies/)
* [Using load shedding to avoid overload](https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/?did=ba_card&trk=ba_card)
* [Dive Deep on considerations and design for Caching](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture/Caching_Dive_Deep/)

#### Accelerated Experiences

Our retail systems take more traffic during peak shopping events such as Prime Day, Black Friday, and Cyber Monday than at any other time of the year. The traditional approach to handling this load is to scale up service fleets. While some amount of fleet scaling is unavoidable, fully scaling fleets to handle traffic for highest-traffic intervals (usually lasting minutes) is neither frugal nor necessary. In advance of intervals with known traffic bursts (such as the first 15-30 minutes of Prime Day), we may choose to proactively put Amazon into an Accelerated Experience (AX) state wherein we serve a lower-fidelity customer experience in exchange for being able to handle increased load at a lower cost. While we have mainly used the AX model for retail peak events, the AX model applies to any number of customer-facing experiences. There are three variations on this model that experience owners can leverage as appropriate for their use case:

* **Reduced functionality.** For example, we might remove Save for Later on the Cart page and reduce the number of results on the Search page. In this case, customer traffic flows through to the experience rendering services, but with reduced workload, freeing CPU capacity to handle more requests. In this model, we recommend experience owners implement programmatic toggles based on traffic levels and CPU utilization, similar to throttling mechanisms. Manual toggles based on OPF headers or G2S2 dynamic configuration also work, but introduce the risk of human error.
* **Edge-cached rendered experiences.** In this case, some percentage of page requests are served from an edge (CDN) cache, clipping the amount of traffic that flows through to the experience rendering services, and subsequently to the associated back-end services. Eligibility for this treatment depends on the amount of dynamic content on the page and tolerance for data staleness. For example, the Homepage is a better candidate than the Checkout page, which has more time-sensitive data such as delivery options. To preserve customer privacy and prevent one customer from seeing another customer’s information on an edge-cached page, edge-cachable page variants must treat customers that receive the treatment as unrecognized, which further degrades the customer experience. We can narrow or broaden the segments of customers that receive edge-cached variants based on factors such as customer membership, depending on the amount of service load we need to clip. Edge-caching solutions require the experience owner to implement a cacheable page variant and configure the [Frontier CDN control plane](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture/) with the cache keys and the segments of customers that will receive the treatment.
* **Client-cached rendered experiences (aka** [**private caching**](https://quip-amazon.com/lCNvAyaBPCzp/Private-Caching-Playbook)**).** This model is similar to edge caching, with the exception that we can retain customer personalization on the customer’s device. The mobile shopping application currently supports private caching, and experience owners will need to set the private caching header and [implement invalidation logic](https://evergreen.corp.amazon.com/cache-invalidation/).

### Context passing

Continuously refetching or rejoining data, even if it's heavily cached, is expensive. It often requires multiple service hops--for example, from VIP -> Service -> distributed cache storage node. You can avoid these costs by fetching once and then "passing" to other service calls, and back to the caller as part of the data payload, be it as extra columns, nested content in a document contract, or something else. This has the advantage of being decentralized and consistent relative to load; it can also save a lot of processing time, but suffers the same kind of usage problems as described in the [Unused computation](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture#HUnusedcomputation) section later in this topic. We have no consideration of whether the data being placed into the object is actually useful anywhere else. Because context passing is often unstructured and not governed, such objects are prone to rapidly bloating, and can lead to significant expansion of [remoting](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Code/#HRemoting) costs and per-request memory requirements. There are some key considerations to work through that can make a significant difference to efficiency when context passing:

* **Resist un-modelled.** While an unstructured “bag” of data may be initially productive, it's a long-term maintenance burden for you and every other service owner in the ecosystem that shares the context. Opt for a common schematic contract and only add entities to it judiciously.
* **Debate/manage collectively.** Ensure that no individual team can add to a shared contract/type without input from the other stakeholders. Ideally, you should always start with data as part of a specific point-to-point or producer/consumer contract, and look for commonalities to consider pulling up to a shared contract or type.
* **Encapsulate and let data drive.** If you can encapsulate behind a contract, and are willing to endure the lowest level of [software maintenance hell (vending libraries)](https://w.amazon.com/bin/view/Tim_Griffith/Thoughts/Software_Levels_Of_Hell/#H4.VendingSoftware), you can potentially operate in mixed mode. Regularly used attribution can be passed explicitly, and infrequently used attribution can still be accessed but will have to refetch locally. This is being used in the consumer stack with [Slate](https://w.amazon.com/bin/view/AmazonAPI/Slate/Design/) and is a hybrid lazy-loading model. It can be powerful, but you'll need to put a lot of trust in the owner/vendor of the library, and you should be actively verifying that it’s behaving as intended.

### Retries

As clients, we issue retries for two reasons:

1. To try to make forward progress rather than failing back to the user.
2. To try to make progress faster, to minimize users’ latency impact.

Making forward progress is the primary consideration since failing will always lead to a bad outcome. At minimum, the client will be kept waiting and get an unsatisfactory outcome; most likely the client will also, at least initially, not accept the failure as an outcome and so retry themselves, thus repeating all the work that was already done and thrown away.

This subject is covered in more detail in the [Services](https://w.amazon.com/bin/view/Services/) wiki article about [Timeouts, Retries and Jitter](https://w.amazon.com/bin/view/Services/TimeoutsAndRetries). However, the efficiency connotations here are important. Acting locally--for example, by retrying your dependencies directly--is the most efficient approach, since it avoids throwing away already-done work. When retries are failing regularly, however, without coordination this can result in excessive waste--for example, by recomputing the subsections of the graph many times to no avail. Therefore, it's important to think holistically about the following key points:

* **Protect your downstream.** Do you understand your modality? If you're getting an abnormally high level of failures, you should reduce your retries, not increase them. An [Adaptive Retry](https://w.amazon.com/bin/view/Jiaqh/CoralRetry/AdaptiveRetry/) policy helps ensure that you don't compound the load on your dependency, without the dependency needing to inform you that it is under load--such as with a throttling [Retry-After](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Retry-After). This form of self-throttling ensures that dependencies don't waste valuable capacity when they're already stressed processing and throttling requests. There is now an out-of-the-box implementation of this strategy for Java service calls included in [CoralRetry](https://code.amazon.com/packages/CoralRetry/trees/mainline-2.0).
* **Communicate to your callers.** Do you provide guidance to your clients as to when they can retry? Providing a [retry-after](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Retry-After) in conjunction with a 429 (too many requests) error if you’re using HTTP, or an equivalent capability for your preferred framework, can help your client understand that you're overloaded and adjust their retry strategy explicitly to accommodate your service’s needs.
* **Work with your originators.** When there is too much load, the best thing to do is make a reasoned decision at the entry point about [what load should be shed](https://w.amazon.com/bin/view/Services/LoadShedding). To do this, it's important to have a good understanding of your traffic paths, such as whether you need tracing. You should then have a feedback channel aligned with the entry point such that they can understand your load and work to protect you and the most important traffic. [Upstream Context](https://code.amazon.com/packages/UpstreamContextJava/trees/mainline) was developed for Java services to carry core control information through a request into a downstream dependency call and is integrated by default in Coral 1.1.

Finally, it's worth understanding normal operation, so that you can identify any redundant work you're doing to accommodate a client's deadlines:

* **Adjust client timeouts.** Can you identify duplicate work from a client? For example, do you have an idempotence token or a marker in your request to illustrate that it’s a retry attempt? If you have a client that's initiating more retries than others, it's possible that its timeouts are set more aggressively than others and don't reflect the distribution latency of your application. By working with such clients, you may be able to reduce redundant work by getting them to increase their timeouts.
* **Be client-timeout aware.** If you exceed a client timeout, the work you're currently doing will be wasted. Is your code aware of the client's timeout deadline? Do you pass that timeout or deadline through to your dependencies as well? If you don't, you may end up doing more work locally, and also initiating more work in your downstream dependencies that is already redundant. To avoid doing unnecessary work and provide another bound on retrying non-valuable work,  make sure that your code checks against deadlines at convenient milestones and propagates deadlines. For example, you might dequeue a request or datum when building an external service call.

### Quality of Service

Quality of Service (QoS) is about enabling your clients to make various tradeoffs that work best for them, rather than providing a single/flat service contract for all. A single or flat contract might align with average requester needs at creation, but it can fail to satisfy the rest effectively in the long run. The Amazon Catalog use funnel chart shown below is a great example. It illustrates that in 2018, out of over 13.6 billion products:

* Only around 6 billion (44%) were buyable
* Only 3.9 billion (29%) were viewed at least once in the last year/trailing twelve months (TTM)
* 99% of all views came from 9% of all items
* 90% of all views came from just 3% of all items

Similar uneven distribution is visible when we zoom in to the head of the catalog. During Prime Day 2018, just the top 22 Amazon Standard Identification Numbers (ASINs) represented more than 4% of all website traffic, and just 267 ASINs represented more than 8% of all website traffic.



The Amazon Catalog use funnel is a classic [Pareto (extremely uneven) distribution](https://en.wikipedia.org/wiki/Pareto_distribution) from both update and read perspectives. Yet Catalog ingestion systems (in 100s) support a single ingestion contract, with an assumption that "All ASINs/updates are equally important to our customers." As a result, ingestion systems scale up and spend more Infrastructure Market Rate (IMR) collectively as catalog updates and ASINs increase over time, but they realize very little to no actual customer benefit. To put this in perspective, Catalog ingestion systems have spent more than $100M on processing and storing updates for items that were never viewed by any customers in 2018.  
  
Your service might be spending lots of IMR for tasks that don't result in any customer benefit. In the face of Pareto distribution, a QoS or tiered contract model might open up extremely large IMR savings opportunities for you. Latency is a common lever for defining a tiered contract. There may be others, such as data quality, data freshness (pre-compute vs. on-demand), etc. As a service owner, you consider adopting a tiered contract that best fits your needs. You can think of a tiered contract in both storage and processing perspectives:

* **Tiered Storage:** Consider tiered storage when your storage costs are high and only a small portion of your data needs to be accessed quickly. For example, Catalog ingestion systems collectively represent 300+ databases owned by 100+ teams and stored using SABLE, DynamoDB, and S3 technologies. Catalog is adopting four storage tiers:
  1. Hot Tier for frequently accessed listings with low-latency access needs.
  2. Cold Tier for infrequently accessed listings with higher-latency access needs.
  3. Archive Tier for listings that have not been buyable for an extended period of time. Archiving is intended to reduce the number of catalog copies from 300+ down to less than 10.
  4. Deleted Tier for listings that have been in an archived state for an extended period of time. Deletion is intended to reduce the number of catalog copies to only a few.  
     Such tiered storage model allows catalog ingestion systems to support 30%-35% year-over-year catalog growth without significantly increasing IMR budgets.
* **Tiered Processing:** Consider tiered processing when not all of your clients require the same flat Service Level Agreements (SLAs) from your service. For example, Catalog ingestion systems are adopting a latency SLA promise based on the importance of an update to our customers. This model ensures that updates against ASINs with extremely high customer demand are processed as fast as possible, while updates against ASINs with little to no customer demand can be delayed for some time. Delaying allows ingestion systems to batch multiple updates against the same entity. Often, batching allows ingestion systems to save IMR by processing only the latest update (and skipping the previous ones).

When defining a tiered contract for your service, you need to first study your clients’ requirements and your service’s cost to serve them. This will help you determine which quality lever(s) are best suited for your system, and how much cheaper you can make your service to your clients while still satisfying their needs. Be extra careful about exposing tiers to your clients without having a proper billing model in place. A free resource very rarely leads to efficient use. You may find out that all your clients demand the highest quality service just because it is free, not because they really need it.

### Limits and feedback loops

Infinitely scaling up, even if you can do so elastically, is not always the right answer. Some systems form feedback loops, such that the more traffic you support, the more additional traffic you get without adding any additional value. A key example of this is the Listings pipeline; as capacity was added, it would immediately get used, but it didn’t result in a commensurate uptick in business activity.

The primary consideration driving traffic is actually an "external repricing engine feedback loop." When two or more sellers are selling the same product, they get matched to a single item (ASIN) and details page, and are bought into competition for the “buy box.” All other factors being equal, the buy box is chosen based on price. This can cause a “penny war” to erupt between the competitive sellers, each adjusting their price down to 1c less than the previous winner in order to win the buy box. While the hope is that this would both improve prices for customers and stabilize at a price 1c below the market price , in reality it does neither. Instead, what occurs is an unstable oscillation. The winner of the buy box will often end up having their margins highly compressed. So, they periodically significantly increase their price to see if they can recapture the buy box while making a higher margin. Maybe their competitors have sold out, or given up, or most damagingly may be slower to reprice... and so the cycle resets. Adding more capacity to the system simply increases the number of entities that can change within a given period, and so triggers more repricing reactions. This loop doesn't add any net value; it simply adds cost. In addition, competitors being slower to reprice can actually harm the end customer by resulting in less competitive prices. This is because the seller with the repricing advantage benefits from the rate of price changes: they will hold the buy box for a longer period than they will lose it for, and when it holds the buy box at the high end of the repricing oscillation, it will make more margin on its sales. There is then very little incentive for the repricing engine to hold the lowest price. A similar behavior was seen with algorithmic trading: [the race to co-locate in the exchange led to less competition and more unfair outcomes](https://www.marketplace.org/2018/09/18/too-much-high-frequency-trading-can-rig-market-says-iex-exchange-founder/), favoring those closer or with lower latency, which inherently is physically constrained to a very small number of participants.  
  
Key considerations for limits are as follows:

* **Understand who you are ultimately serving, such as machine or human.** Humans naturally can't handle many transactions per second and don't take to errors or waiting well. Machines can handle many hundreds or thousands of transactions per second (TPS), and can be made very patient and tenacious with handling errors. The better you know who you're ultimately serving, the more informed and granular decisions you can make as to whether their usage is appropriate or excessive.
* **Understand the mechanics of the request.** Will a response simply trigger a new request? And if so, what's the marginal utility, or increase in business value, of each request? Consider balancing the value against the estimated cost of conducting the operation, so that you don't conduct net-negative work.
* **Understand the latency sensitivity of the request.** If the processing is delayed for a few milliseconds, what would happen? What about a second? minute? hour? day? week? Could your system process the request more efficiently if it was to delay its processing? For example, what if you amortize it with other requests, such as batching updates to an entity? This is often only available to asynchronous systems; however, in some circumstances synchronous systems can benefit from micro-batching, such as using a [buffer](http://reactivex.io/documentation/operators/buffer.html) on a reactive stream.

An immediate consideration is to [throttle](https://w.amazon.com/bin/view/Services/Throttling) work, which is great for protecting a service and ensuring fairness. However, throttling often doesn't make the work go away; it just sits upstream--for example, while the user clicks retry in a browser, or a client service waits before retrying. If you have the capability to provide queuing internally, and can realize benefits from deferring processing for a period of time, you might be able to realize considerable efficiency savings.

## Unused computation

Doing work on demand ensures that work is only done as required. However, doing so is not always feasible, in particular when a human is waiting for the outcome and your service  needs to be user responsive. In many situations, doing the work on demand can also be significantly less efficient that doing it in a bulk/throughput-oriented fashion for a variety of reasons, including I/O and compute economics, the ability to do ahead-of-time planning and optimization, and the potential to schedule processing at off-peak hours.

The two primary behaviors that lead to unused computation are:

1. Pre-compute, where computation is done either on a schedule or in response to input or dependency data changing, completely agnostic of consumer request.
2. Pre-fetch, where the calling client or consumption path is initiated preemptively on behalf of the client--for example, before the end client themselves decides they need to invoke. This lets you furnish the result with much lower latency when the client decides they need it.

### Pre-compute

Pre-computation involves proactively performing work to materialize and stage a dataset ahead of time, so that clients can just read the dataset rather than wait for an on-demand computation. The [Data Warehouse](https://w.amazon.com/bin/view/Data_Warehouse/) and [DataPath Precompute](https://w.amazon.com/bin/view/Datapath/Precompute) (Reactors) are two significant examples of this pattern; Data Warehouse works at a whole-dataset granularity, and Datapath works at a per-datum granularity.

Pre-computation platforms often form dependency graphs. For example, the Offer and Remote Catalog Services (ORCS) join /ols/Listing to catalog/xref to produce buy/listing, and then join buy/listing to availability/fms and availability/gpi in order to form buy/buy. When a datum in the input scope changes, the join is recomputed and stored, which in turn triggers the next function in the graph.

Although pre-computation is often much more economical to run per datum, pre-computing a value that is never used wastes resources. In the consumer website, the "never used" consideration isn't just common--it's actually the majority case, as shown in the [Amazon Catalog use funnel example](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture/#HQualityofService) from the [catalog pipeline efficiency white paper](https://share.amazon.com/sites/IOP/Document%20Repository/Projects/Catalog%20Efficiency/Efficiency%20Opportunities%20in%20Catalog%20Ingestion%20Pipeline.docx).

So, when doing pre-computation, the critical consideration is "what percentage of the work that was precomputed ever used?" This often can be broken down into progressively more granular details:

1. Rate of change over time
2. Datums within a dataset
3. Fields/attribution within a datum

Measuring usage can be hard, especially if you vend the data rather than rely on callbacks and so don't have line of sight into client usage. However, these measures are critical to both your and your clients' efficiency. Working with your clients to understand what they need and how they are going to use the data lets you constrain the pre-computation to only what will be used. And setting these expectations as early in the pipeline as possible can significantly improve both your generation efficiency (less to compute, less to write) and your clients' consumption and processing efficiency (less to read, less to process). Using a platform that can do the impact analysis and management for you is even better. [Cradle](https://w.amazon.com/bin/view/BDT/Products/Cradle/Overview/) and Datapath already have some optimizations present and are continuing to invest in this space to reduce unnecessary wasteful computation.

### Pre-fetch

Pre-fetching involves anticipating a client call and starting it early, before the client has explicitly requested it, to reduce the client's perceived latency. For example, if fetching a customer’s cart takes 3 seconds from when the customer clicks the checkout button, that 3 second wait time results in a poorly perceived customer experience. But if the browser code preemptively fetches the cart when the customer moves the cursor towards the checkout button, then by the time the customer has centered and clicked it might be able to save a second or so off of the perceived latency, a significant CX win. Of course, if the client stops short, or clicks a different nearby link, the cart-fetching work that was initiated is wasted. So, much as with pre-compute, measuring how often the pre-fetched call is used is critical. Measurement allows us to understand the tradeoffs between realizing customer experience/latency benefits and materializing waste, so that we can find an appropriate balance. Measurement is often difficult. Frequently, the pre-fetch won't be just the client calling and holding the response, but rather issuing the pre-fetch as a “fire and forget”, relying on the underlying services to not only compute the result but also cache it for a period of time.

## Redundant capability

Virtually nothing will beat the cost-effectiveness of not running a service. Assuming that a service is  needed, economies of scale generally favor having only 1 version of the service serving all clients, rather than many. Ask yourself: Why can't I just turn this off? and Why can't I consolidate my clients to another existing service? Answers to these questions help identify who you serve and how you serve them. They also identify other services that share commonality and so may be able to share everything from learnings through to coalescing execution and clients. This often requires taking a bigger-picture view of the following questions:

* **Customers:** Who are you serving? or Who would you like to be serving?
* **Domain and purpose:** What wider application sphere(s) is your service tailored towards?
* **Similar offerings:** What planned or existing and former services have similar mechanics to yours?
* **Incentives:** How do you ensure your clients are making the right value tradeoffs in how they use your service? Likewise, how do your dependencies ensure the same of you?

### Customers

Who are the customers of this service? It's important to know who your customers are and maintain a relationship with them to understand their needs. In good operation you'll want to serve all your customers well, of course, but it can be important for both availability and efficiency reasons to understand the relative criticality of your customers and how to identify the segmentation. If you end up in a situation where you cannot serve every unit of work, you should drop the units of work that are holistically least valuable first. Some key questions to consider are:

* **Which of your customers are latency sensitive?** Those that are most sensitive will be less eligible to traffic shaping by throttling or  deferring till later, because throttling those customer requests will either quickly result in overall failure or cause aggressive retry policies that can exacerbate overall system load. Understanding what your customers’ latency requirements are and codifying them into a Service Level Agreement (SLA) that can be monitored helps you to be proactive in identifying and mediating customer pain, and can help mitigate expending effort on work that will never be useful.
* **What grades of customers do you serve?** An external customer will generally be more important and have tighter considerations than an internal customer. Likewise, serving a real human, like a customer, will generally be a priority over serving a machine task such as Robot/Web Crawler traffic. The key to determining criticality is understanding what the adverse side effects will be if you fail or delay serving a particular customer request.  
  Finally, having a good working relationship with your customers is also beneficial in rolling out changes in how your service is invoked, such as a change of protocol, authentication mechanism, or endpoint location. This can help you quickly realize efficiency wins, since simply providing the support for a new, more efficient serialization protocol alone achieves nothing. You will need your clients to migrate in order to realize the savings, in both your service and theirs.

### Domain and purpose

What does this service do? What kind of domain does it serve—for example, a  website customer request? What kinds of data does it work with, like catalog items and offers, payment instrument, or customer? What unique business value does it provide? Having a clear statement of where the service fits into the wider architecture is not just useful for onboarding new team members; it drives clarity of thought around what your service should be doing and allows for comparison against what it actually does. This allows near neighbors to understand your service and vice versa, and forms the foundation of being able to work together to optimize the architecture holistically. For example, suppose that several services in the same domain, doing near-identical operations from the same dependencies and data sources, are always being called for the same set of requests. Each of these services can be locally optimal; however, globally the cost of doing the work is distinctly sub-optimal. With greater awareness, better division of labor and informed composition can follow to provide a more globally optimal and efficient solution.

### Similar offerings

What other services are you aware of that serve similar customers, operate in similar domains, and provide similar functions or business value? The side effect of Amazon's strong focus on Ownership and Bias for Action as well as the [2 Pizza Team (2PT)](https://improvement-ninjas.amazon.com/s3files/s3get.cgi/RevisedPizzaDoc.doc) structure mean that it's common to find that there are 2 or more services that effectively serve the same purpose. There will always be some variations in implementation of services that perform identical functions. These variations often stem from a partitioning of the domain. For example, one service that does Foo is for Vendors, while the other service does the same for 3P Sellers. Identifying those services that are very similar and establishing relationships with the other team(s) provides a series of opportunities:

* **Can you share learnings?** You will have succeeded and struggled on capabilities that other teams are just starting to consider (and vice versa). Sharing insights on what worked and what didn't can significantly reduce risks for the team about to build, and can result in a superior solution that’s actually a ”2nd version” build, taking all the good and trying new approaches for the bad from the first build.
* **Can you share components?** Are there commonalities that would better serve all teams by abstracting/generalizing and breaking out into a separate or shared offering? This can allow for removing undifferentiated work and can have an outsized effect.
* **Can you consolidate offerings?** Is it possible the services and/or the domain they are in can be aligned or generalized such that the offerings converge into a single “best of breed” offering? Note:  Don’t propose building a new service to unify them all; as that just [increases the number of competitive offerings](https://w.amazon.com/bin/view/Tim_Griffith/Thoughts/Software_Levels_Of_Hell/#H1.Findandusesomeoneelse27s). Instead, focus on identifying the existing service fittest for the purpose and looking at how to extend it to cover the expanded domain. For example, the Listing pipelines were different for Vendors and 3P Sellers, feeding into Retail and Marketplace processing pipelines respectively. The Brooklyn program determined that the 3P Seller modeling and Marketplace processing pipeline was the best fit for convergence, and so worked to represent Vendors as 3P Sellers and to start processing Vendors' data through the Marketplace pipeline.

### Incentives

What is the cost per business unit of value served by this service? How do you make these costs visible and/or allocate back to your customers? If you're not making your service's cost visible to your clients, then they will likely operate on the assumption that your service is free, which creates the wrong incentives. For example, a client might find it cheaper to not clean up obsolete data or delete a scheduled report, because they only see the costs to their own time of making the change, and not the accruing costs of these useless obligations that your system is honoring on their behalf. Tools such as [Osmium](https://w.amazon.com/bin/view/Osmium/) can be used to set up a metering, allocation,  or billing model based on your existing metrics and infrastructure bill, and is highly recommended to help align your priorities. Getting the billing model right is difficult; too simple and you can miss aligning your clients to one or more of your key cost drivers, but too complex and you'll have client confusion and frustration. But just because it's hard to get right doesn't mean you shouldn't do it. Don't let the perfect be the enemy of the good; get a billing model out, pilot it, and iterate/improve it.

# Caching

Caching is a technique to reduce effort by reusing previous results.

## Immutability

Caching immutable data is simpler than caching mutable data because you do not have to worry whether you have the latest copy. You have the current version if it’s in the cache. A common technique when caching data that could change is to have a mutable pointer to an immutable business object, like a snapshot. For example, in Apollo, there is a "latest" pointer (mutable) to a specific deployment (immutable). This specific deployment contains a list of package versions. Each package version contains immutable build artifacts (jars, brazil config) that can be safely cached. In addition, Apollo will reuse these cached artifacts across environments on the same host, providing faster, consistent deployments.

Another example proposes that Netflix and Comcast could save money by pushing (immutable) movie content closer to their consumers, perhaps by using Content Distribution Networks like Amazon’s CloudFront. When a Netflix subscriber clicks **Play**, Netflix could redirect the subscriber's browser to a geographically close (immutable) copy of the movie.

## Request-level consistency

Caching within a request scope can provide consistency. For example, Gurupa is a website rendering platform where multiple teams write code to build the detail page. Every portlet displaying the price will make the same remote service call. Website users would be confused if the price was listed more than once and was not the same everywhere. The Gurupa platform will deduplicate remote service calls and store the result in a request-scoped cache so a product's price is consistent across a page.

## Data modeling

Cache key selection determines correctness and effectiveness. Under-specifying a cache key may return the wrong result. For example, a cache key that contains URL but not language may return a French webpage instead of an English webpage. Overspecifying a cache key may reduce the cache hit rate. For example, including a request Id in the cache key would be counterproductive.

Don’t try to do too much work within the request scope. Service owners sometimes assume cache reads are free. But you should think of a cache operation as a remote service call, because there is latency and serialization overhead. Worse, there is variable overhead because you don’t know what caching tier your data is in, or if your data is in the cache at all. As a rule, your data model should minimize the number of remote service calls within a request to no more than ten. To get around this limitation, you may be able to calculate results offline so they are ready to use later. For example, we rebuild search indices periodically so we don’t have to iterate the entire catalog when our customers are searching for content. Another example is the way Google precomputes results for the billion most queried terms, so that search results are fast.

Another caching practice to avoid is fan-out. A fan-out is when the data model requires data from multiple cache partitions, or sometimes all of them. A server will send parallel requests to several cache partitions requesting data. As caches are quick, there is a temporary network traffic jam as all the responses come back around the same time. In addition, data models that are fan-outs are less partition-tolerant, since a single cache partition outage will break a non-proportional amount of traffic.

Related to fan-out, a fan-in is when most requests rely on a single record in the cache. This is bad as there can be much more traffic to a single partition compared to the other partitions. This partition is an availability risk and the caching fleet has to be scaled to accommodate it, which is not a frugal practice. Fan-ins are less partition tolerant, because losing that partition has a higher blast radius than losing other partitions. We see this with runtime configuration or hierarchical data sets.

Caches can contain special markers that indicate the absence of data. For example, you can configure Java to cache DNS entries (200) and to cache failed (404) lookups. This negative cache can help prevent you from overwhelming downstream services when your clients request something that does not exist. You typically have different policies for negative caches, such as a shorter Time to Live (TTLs), if you are expecting the record to eventually show up. Note that you would not cache all responses: for example, caching a permission denied does not make sense.

## Removal

Caches should have different replacement policies for immutable and mutable data. A cache should replace immutable cache entries when the cache is full, and have some upper limit to how long the entry can be in a cache. We want that upper limit because maybe we didn't get the immutable value right in the first place or the data is corrupted in the cache. A reasonable upper limit is situation-dependent, but an hour seems a decent max value.

A cache will replace mutable data when the cache is full or when the caching policy states the entry is invalid, usually by using duration (TTL). A best practice is to let the data provider set how long a business object can be cached for, since they understand the business best. The HTTP specification allows the server to specify how long the client should cache an entry for, and what the cache key is. For example, we would want a different caching policy for the weight of a product, which doesn’t change, compared to how to calculate tax, because there are legal ramifications if that data is wrong or stale. A data consumer will probably get the caching policy wrong because they do not understand the business.

Cache invalidation is when we remove an entry from a cache out of band. Operators of services MUST be able to administratively remove entries from their cache when the business demands. For example, if the retail website is displaying illegal content, we update the authority in the product catalog. Then we need operators in the services between our customers and the product catalog to invalidate cache entries for that particular offer.

## Runtime

Cache hit rate helps you understand how effective your cache is. A cache hit means the cache returned a record for the given key. A cache miss means there was no record for that key. Putting only one type of data into a cache will help you monitor and scale your use cases independently. Use cache hit rate to estimate how much less work your fleet is doing to calculate whether your cache has an appropriate return on investment. However, note that there are some services that do not make sense to cache. For example, it might be cheaper to just calculate a result than to leverage a cache.

An important feature for caches is an efficient way to determine what entries will not be helpful in the future, so that they can be replaced with new data as the cache fills up. Most cache vendors support LRU (Least Recently Used) as a cache replacement algorithm because it is fast and easy to implement. Sometimes more advanced replacement algorithms are required. For example, Sable uses MQ, which does a better job protecting against attacks like full table scans, and also does a better job cheaply selecting less frequently used entries for replacement. You can build a simple simulator to replay your logs and determine whether you are using the correct cache replacement algorithm. The outcome might be that you need to change your caching configuration, request additional features from your caching vendor, or switch to a vendor with a better replacement algorithm.

Caches have different types of benefits depending on where they are within a service-oriented architecture. A cache hit on a service closer to the customer will save more work than a cache hit on a leaf service. However, as services closer to the customer tend to contain more specific results, such as a personalized offer, the hit rate is lower. Caches in the middle of the service-oriented architecture provide most of the value, such as a non-personalized offer. Traffic to leaf services like Sable benefit less (if at all) from caching, because upstream caches have already filtered out the repeated traffic.

Caches can be client side or server side. Client-side caches can eliminate the need for a remote service call to the server. Server-side caches require the client to make the remote service call, but don't require the client to add a cache. In addition, the server can be more selective on cache key selection or cache intermediate results before joining with cache-busting keys. This technique is called push down.  
  
Caches can be local or remote. A local cache can be in-memory (L0), in-memory off heap (L0.5), or in a separate process on the same box (L1). In-memory caches can reduce serialization costs. Local caches in separate processes can reduce GC pauses. A remote shared cache (L2) allows a fleet to cache a much larger dataset and has better consistency. An L1/L2 cache configuration is common, where a program will look in the L1 cache (miss), then the L2 cache (miss), make a remote service call, write the L2 cache entry, write the L1 cache entry (with jitter on the TTL), and return the result. When the L2 contains a cache entry, the program will write the value to the L1 cache using time remaining in the L2 cache (jittered) as the TTL. We use jitter by subtracting a random amount so that all cache entries do not expire at the same time across the fleet.

Caches have latency and CPU overhead to them. A common best practice when calling a remote service is to start a timer, make the call, and stop the timer when you get a response. Create latency metrics keyed by return code and add alarms for them so that you know how much of your request time budget is spent reading from caches, including which tier. You also want to know when you have a shift in traffic, for example, if you have a lower cache hit rate or if the payload size shifts. You need to review your profiler results to understand the overhead of serialization.

Caches need to be properly sized, with the right number of records or bytes to work correctly. A common failure mode for caches that are too small forces you to fall back to more expensive tiers. An extreme failure mode for caches that are too small is we replace useful entries with entries that are less useful. Each cache tier can have a different size. L0 caches are typically smaller than L1 caches, and L1 caches are typically smaller than L2 caches. To determine the correct size you need to understand the size of the data hotset. To determine whether caches are too small, you can build a dashboard for each of the tiers that identifies why the cache is replacing entries. Replacing entries because the entry has expired, which is mutable with TTL, is fine. Replacing entries because you are out of space may mean you need to increase the size of the cache. You can build a simple histogram by parsing your logs. Ideally you size the hotset so it fits in your L0/L1. You size the L2 to maximize your ROI.

All non-trivial L2 cache setups are partitioned. Each partition contains a subset of the cached dataset. This partitioning allows you to increase dataset throughput and dataset size, but throughput to any single record has finite throughput. A decent cache vendor will not reshuffle your data when you take a host in or out of service, causing a spike of cache misses.

An L2 cache is a remote service. Just like any other remote service call, your service can overwhelm an L2 cache. You may need to rate limit calls to the cache, for example, by adding a circuit breaker and removing retries. As L2 caches scale well at the dataset level but not at the record level, you may want to serialize or de-duplicate inflight reads to the same record.

## Safety controls

As a general principle, we use caches as a latency and cost optimization but do not require them for availability. Consider NOT using caches as your architecture becomes simpler and more predictable. This might not be practical, because Amazon is addicted to the latency, cost, and pseudo-availability wins when using caches. Here are some ways you can protect yourself and your downstream services from a retry storm when your service gets slower due to cold caches:

* **Add crush tests to your pipeline.** The crush tests should send traffic to your service so that it will break your caches. A successful crush test will demonstrate that your service will recover when traffic returns to normal. Repeat the process with your caches disabled.
* **Periodically turn off your caches in production in the middle of the night.** This will let you know how your clients will handle the increased latency, and you can easily turn the caches back on if the website goes down.
* **Add circuit breakers to your remote service calls.** It’s better to accept a temporary outage than the possibility of a sustained outage. This includes calls to your L2 cache. For example, you can imagine decorating Guava’s cache provider with a circuit breaker.
* **Use speculative invalidation, at a request interval you can choose.** Speculative invalidation will periodically  force a cache miss by skipping the check if there is an entry in the cache. This alternate code path will still update the cache with the result. The impact is backend services will receive a predictable amount of jittered traffic for the most frequently requested items, yet the cache entries will be fresh.
* **Leverage soft and hard TTLs.** The HTTP RFC defines communication on how a server and client can negotiate cache policy for a record. An expired soft TTL means a client can still use a cache entry, but should try to refresh the value. An expired hard TTL means a client MUST NOT use a cache entry.

## What should you know about your use of caching?

You should be able to answer the following questions if you are using a cache:

* What metric and alarm shows that there is sufficient return on investment for my cache setup? This should be a function of cache hit rate for each business object you are caching.
* How do I certify my cache is scaled for a high-velocity event? Hint: You need to know what the traffic signature will look like, not just a scalar traffic number
* Did I set the object size and capacity size  correctly for my L0, L1, or L2 caches?
* Am I using the correct host type for my cache fleet?
* Do I understand traffic patterns sufficiently well to know that there are no other places in my architecture that could take advantage of caching?

**Metrics**

## Metrics and measures

I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

*Lord Kelvin, "Electrical Units of Measurement" (3 May 1883)*

Amazon uses metrics and measures to understand the previous and current behaviors of its systems.  Metrics are the methods by which we measure ourselves. Some metrics are standardized within an industry or within Amazon.  Others are customized for understanding a very specific behavior.  We measure our software, hardware, operations, etc., along many dimensions, such as performance, reliability, security, efficiency, cost, business value, and customer experience. And we make these measurements visible to a wide variety of interested consumers--for example, in order to improve debugging and investigation time for other service owners. Potential consumers of your measurements include software developers, technical leaders, on-calls, UI designers, data scientists, call leaders, hardware engineers, business teams, and automated mechanisms.

## What to measure

When you're choosing the right metrics to track for your service, think about both your customers' expectations and your own. Make sure you can characterize and track normal and abnormal behavior. This includes not only the low-level details of your service's behavior, but also any functional business metrics that reflect the higher-level experiences provided by your service.

### Track what your customers depend on

At a minimum, your customers typically include:

* your team
* technical and managerial leadership
* TOS and call leaders
* the teams owning services that call your service or that you call

You should also think about customers that might have zero knowledge about your service but happen to end up looking at your metrics (for example, via your dashboard) as they drill down through the service stack(s) your service is part of. Work backwards from your customers to understand what they depend on; put yourself in their shoes. What metrics are your key indicators of health, performance, and level of service? What constitutes different levels of service? Have you made your key metrics easy to understand?

#### Measure availability and latency

Almost all services should be monitoring at least the two metrics that represent **availability** and **latency** for their service.

* **Availability (success rate)** measures the fraction of valid units of work that the service receives and processes successfully. For a request-based service, the units of work are requests.  For event- or task-based services, the units of work are events or tasks.  It's common for our most critical services to support success rates in the "four or five 9's" (i.e., 99.99% or 99.999%).  Maintain a separate metric for invalid requests/events/tasks that your service receives.
* **Latency** measures the time it takes for the service to process a unit of work, whether it's a request, event, or task. For event- or task-based services that use an ingress queue, include the time spent in that queue when tracking the service's overall latency. This means measuring both the processing time from the original task submission by the client through to task completion as well as the portion of time spent in the queue. Be especially wary of "hidden queues," where requests wait before your service is able to mark their arrival time, such as within a TCP receive queue where only the OS knows that requests have arrived.

For offline or batch activity, the above metrics are still important, but you may also need an indication of throughput, the rate at which units of work are processed.

For more information, see [CRE Guidance on Availability Metric Measurement](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance/#HAvailabilityMetricMeasurement).

#### Measure per API, per event/task type, and per request/event/task "size"

We recommend measuring both availability and latency on a per-API basis for request-based services, or a per-event or per-task type basis for event/task-based services. Consider including pieces of the caller's identity as dimensions in the metric. These might include service name, calling API, region, user experience/page, etc. This makes it easier to locate and diagnose issues and keeps critical APIs and event types with small traffic volume from getting drowned out. Similarly, if there are different amounts of resources (compute, memory, network, storage) required for different "sizes" of requests/events/tasks (e.g., number of items in a shopper's cart), this affects metric expectations, alarm thresholds, anomaly detection, and other metric analyses. This doesn't mean capturing separate metrics for every possible size of activity, but rather something along the lines of categorizing activity into small/medium/large groups. That being said, the cost of expanding the number of metrics (by API, type, or size) should also be taken into account for performance and storage reasons.

#### Don't forget to track what *doesn't* arrive

Problems in the systems that support ingress of requests or tasks to your service (such as load balancer spillovers or queuing system issues) require separate monitoring. These issues can prevent the service from receiving request/events/tasks, so they might not be evident in the service's own metrics.

#### Don't forget to track what *doesn't* complete

Some metrics are based on request/event/task completions. If a request never completes--for example, it may never increment either a success or a failure counter--the request's total time of processing will never be recorded. Not only does this distort the total picture of service behavior, but it also makes diagnosis much more difficult.

#### Ask yourself what else your customers care about

Try to identify other aspects of your service that your customers care about, and define metrics around them that help you measure the service's behavior. For example, can the responses from the service vary in quality or completeness? Can you measure and monitor that?

### Track the host-level resources that you depend on

Common metrics you should measure and monitor:

* CPU, memory, network, and disk utilization on your hosts.
  + Memory utilization can be tricky to interpret. Ideally, most of the memory on a host is occupied at any given time, since you want to hit in a memory cache of data rather than going to storage. Therefore, you may have to look at the metrics for different categories of memory usage.
  + For disk utilization, measure both the bandwidth consumed and capacity used. An all-too-common failure is for local disks to run out of space due to excessive logging during failure scenarios. Alarms should fire well in advance of such events (see the guidance on Alarms below).
* Cache latencies and hit rates -- Scaling your hosts up to full performance often depends on cache hit rates. Services with high cache hit rates are sensitive to cold-start issues and cache-busting queries. Throttling strategies may consider throttling clients whose requests are showing inferior hit rates and therefore consume more resources.
* Time waiting in queues and total size of queues -- Increased queuing may be associated with delayed processing or errors that cause reprocessing. If dead-letter queues are used, monitor their size. If the service supports variable-sized messages, monitor the message sizes, as they impact processing times.

### Track the services that you depend on

Common metrics you should measure and monitor:

* Availability and latency of your own service's calls to its dependencies -- Keep in mind that your service's view of these metrics might not be apparent in the dependency's own metrics. Your service might represent too small a fraction of calls to a dependent service to trigger its alarms or even show noticeably on its dashboard. Or your service calls might not be reaching the dependency at all, or they may be delayed due to issues in some networking or connectivity component.
* When incomplete, erroneous, or missing responses from dependencies (such as HTTP 4xx's and 5xx's) result in giving up on satisfying a request to the customer, you must track these as "[fatals](https://w.amazon.com/bin/view/RealTimeLogAnalysis/Customers/FatalsDefinitions)".
  + You must generate only one fatal per request.
* Latency of database (DB) queries -- Databases are another type of dependency. Changes in latency can be tied to DB-side query plans shifting or missing an index (combined with row growth), or an inefficient query deployment.

Observing the relationships between your service's metrics and your dependencies' metrics can provide powerful insights, such as identifying the reasons for varying "wait times" being experienced when you make calls to other services.

### Track client behaviors

The scaling and configuration of a service generally depends on assumptions that were made about the traffic it receives.  Such assumptions include not only the total volume over time (such as diurnal or peak event variation), but also the distribution of specific operations, their processing costs, and their latencies. You should track changes in patterns of client requests (or events/tasks submitted by your clients) because such changes can cause issues for your services.

For each API, event, or task type that your service processes, track your service's incoming traffic by client (and possibly by client per API -- see above). For example, track the top 10 callers both by latency and call volume, and make sure these are on your dashboard to easily identify callers that may be overloading your service.  Working back from the customer, track any other metrics that help quantify the behavior/quality of service you are providing for each client.  This includes tracking any types of "fatal" responses that your service returns to each client.

### Track key internal metrics

Track key metrics that help you understand the internal health of your own service and to detect signs of trouble early. Once an incident has surfaced, perusing logs is painful and time consuming, contributing to longer resolution times. Internal metrics give visibility into what is going on without diving into logs.

Common metrics that services should track:

* Java Heap Usage and garbage collection (GC) metrics -- Tracking heap usage after each GC can alert you to memory leaks. High GC counts per unit time can signal problematic object creation and heap usage patterns.
* Utilization levels of internal application resources such as file descriptors, connections, and threads -- especially when the resources are maintained in pools.
* Errors with specific type or locality info -- Generic counts of errors or fatals require the on-call dev to examine logs to fully understand what is happening. Specific error metrics by type and context can avoid such diagnostic hunts.

## Monitoring: Automate alarms around your metrics

Characterize the normal behavior of your metrics, and set up alarms to catch abnormalities.

### General monitoring guidance

* Have alarms on the key performance indicators (KPIs) for your service. Generally, this includes latency and availability at a minimum, measured as a fraction of valid requests successfully served within Service Level Agreement (SLA). Include these as dependent metrics in your deployment auto-rollback monitors.
* For workloads that dynamically (and predictably) fluctuate in intensity, monitors are particularly difficult to set up. For example, many Consumer, Digital, and Other (CDO) services experience diurnal variation in traffic load and request mix. If the high/low thresholds are set far apart, in order to account for the expected variation, you will be slow to catch abnormal behaviors when your load is in the "middle" of the threshold range.  If you set them too close together, you will generate false alarms at the peaks and troughs.
* On-calls should be notified well before your customers/clients start complaining. Ideally, alarms should fire in time such that quick issue resolution can prevent perceivable impact.
  + Use 1-minute metric capture intervals, and alarm after 3-5 successive data points are in breach of the alarm threshold. Set your alarm thresholds as tight as possible without producing a rate of false positives that burns out on-calls and decreases alarm confidence. This means tolerating a small number of false alarms (perhaps, a few per year). If you never get a false alarm, you should double-check that your thresholds are tight enough.
* Measure and alarm on the observed latency and availability metrics your service sees from its dependencies, preferably at the API level for critical dependency calls.
* Measure and alarm on at least the following local-resource metrics: percentages of CPU utilization, memory utilization, and disk space utilization.  We also suggest monitoring disk bandwidth/IOPS utilization.
* For Java-based apps and other applications that allocate and manage their own heaps, measure the rate of GC, the usage of the heap, and the size of the heap after any GC.  For Java, you can use metrics such as JMX CollectionCount, CollectionTime, HeapMemoryUse, and HeapMemoryAfterGCUse.
* Define your monitors using code and deploy and test them through your pipeline.

### Alarming properly

When a monitor fires, you want the resulting alarm action to drive the most effective response.

The typical alarm output is in the form of an auto-cut ticket. Keep the following in mind when setting up such a ticket:

* Cut the ticket at the proper severity. Use SEV-2 if it needs rapid attention.
* Use the right C/T/I. Your team (or your dependency teams) may have multiple C/T/I's.
* The message in the ticket is important. When designing it, imagine that the reader is an on their first rotation as an on-call. The ticket text should:
  + Clearly **describe the context, meaning**, **and relevant metric values** of the alarm.
  + **Link to the response runbook** **section** for that specific alarm. Don't just point to a giant wiki of procedures. Point the resolver right where they need to go.
  + **Link to the right dashboard(s)** to support diagnosis.

## Dashboards

Get your metrics and threshold levels onto dashboards. Put the key health metrics at the top of the dashboard, including clear and concise definitions, sampling details, and alarm thresholds. Follow the [Dashboarding Best Practices](https://w.amazon.com/bin/view/Dashboarding/).

Try to design your dashboard to support at least these two important activities:

* Quick and confident identification of service health -- Whether during daily on-call monitoring or when paged into a site-wide (SEV-1) event, an on-call should be able to instantly gauge the health of their service at all times.
* Issue localization -- When responding to an incident, the dashboard should facilitate identification of the issue source(s) as well as some insight into what might be going on.

## Weekly metric reviews

Teams should run weekly metric and dashboard reviews (within their teams) to identify metric behaviors that may require attention. This can highlight missing alarms and identify gradual trends (day-over-day, WoW, MoM, YoY) that could eventually cause issues. If you identify potential issues proactively, you prevent customer impact and you reduce on-call pain, The effects of new features, especially if they are slowly adopted, can sneak up on you.  Observing metric trends over time can also lead to insights that help decision-making, scaling, and planning development work.

On-calls should relay any difficulties they had during their shift -- in utilizing either the dashboard or the metrics to quickly diagnose issues (or handle false positives).

## Study your metrics using histograms

Rather than looking at only a few fixed percentiles, we encourage developers to understand the full distribution of important metrics using histograms. These often reveal unexpected aspects of the metric’s distribution, such as clumping around certain values, multiple modes, or gaps. Diving deep to understand what is causing these phenomena provides insights into opportunities to improve behaviors for customers.

## Use trimmed means to avoid noisy averages

Setting targets at specific percentiles can actually encourage teams to make changes that result in clumping around those specific target values.  Meanwhile, relying on expected values (averages) can leave your goal metrics vulnerable to noisy skew by a few high-percentile outliers. Trimmed means, which are averages calculated after excluding high-percentile outliers, address these issues. For example, the tm99 is the mean after trimming values beyond the 99th percentile. We recommend trimmed mean metrics, such as tm99, for goals on metrics like latency, where the you want to improve the average behavior for customers but remain insulated from outlier noise.

## More resources

* [Operational Excellence Online Course on Metrics](https://w.amazon.com/bin/view/EE/Learn/OE/Metrics/Part1/) -- Great source for the "How" and tooling tips accomplish the above best practices
* [CRE Guidance on Monitoring and Alarm Thresholds](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance/#HMonitoringandAlarmThresholds)
* [Monitoring Wiki](https://w.amazon.com/bin/view/Monitoring/)
* [CRE Guidance on Availability Metric Measurement](https://w.amazon.com/bin/view/Consumer/ReliabilityEngineering/Guidance/#HAvailabilityMetricMeasurement)
* [Dashboarding Best Practices](https://w.amazon.com/bin/view/Dashboarding/)
* [Old BSF ServiceMetric](https://w.amazon.com/bin/view/ServiceMetrics/)
* The [CloudWatch Metrics Histogram tool](https://cloudwatch-console-tools.iad.amazon.com/histogram.html) and [its wiki page](https://w.amazon.com/bin/view/Histogram/)
* [Wikipedia’s definition of a Histogram](https://en.wikipedia.org/wiki/Histogram)
* [Trimmed mean](https://w.amazon.com/bin/view/TrimmedMean/)
* [Wikipedia’s definition of a Truncated mean or trimmed mean](https://en.wikipedia.org/wiki/Truncated_mean) -- note that the common trimmed mean trims symmetrically at both ends. In many applications, like measuring latency, the small-value low-percentile values can contain valuable information. Also, because they are small, they pose less risk of skewing the average.
* [Trimmed Mean: A better latency Goal; Histograms: Better Latency Visualization](https://broadcast.amazon.com/videos/275181) -- This POA talk covers both the histogram and trimmed mean topics well

**Scaling**

## Scalability

We scale software within the Consumer business in two ways:

* **Load scaling,** our ability to accommodate more users/load on our software
* **Functional scaling,** our ability to add new features quickly to delight customers

Scaling your software is always a trade-off between being efficient, being available, being scaled, and being extensible. Scaling requires thinking about patterns to gracefully degrade or shed traffic and prevent a system from failing when the rate of incoming requests exceeds the rate that the system can handle effectively. Scaling requires understanding how your system scales, but also, critically, how it doesn’t scale. No software is perfect; you are always making a trade-off. It is better to be explicit about the trade-offs than have hidden scaling bottlenecks.

Load scaling requires understanding the dimensions on which your software scales so that you can be [efficient](https://w.amazon.com/bin/view/CEH/Efficiency/). Note that scaling isn’t always about scaling for more and more requests. For example, Amazon’s promise calculation systems scale linearly with the number of requests, but exponentially with the number of fulfillment centers. This isn’t a big deal when the number of fulfillment centers is small, but as the company pushes for more, smaller fulfillment centers, this is an enormous bottleneck – one that is far more important to address than the linear nature of per-request scaling.

The golden rule is that you should plan for your system to sustainably scale to 100x – whether that means 100x the traffic or 100x the number of use cases, and for approximately 5 years. If your system successfully meets these criteria, it’s okay if it eventually needs to be replaced. At that point, the business has changed enough that assumptions are no longer valid and a holistic re-evaluation is warranted. Using this rule avoids thinking too far in the future about “what if?” scenarios, and helps you focus on the practical problems of the next few years.

## Load scaling

To effectively scale for load, you need to first understand what factors your system scales on. Within Amazon’s Consumer business, most systems scale based on CPU, memory, network throughput, or disk space (or some combination of these). Using tools such as [profiling](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Code/#HProfiling) can help identify hot spots in your code, and aren’t just about identifying cost reductions but can also help with scalability. While you can estimate which factors are the primary scaling factors for your system, it’s most accurate to test. At a minimum, you should test [yearly as part of peak prep](https://w.amazon.com/bin/view/CEH/Availability/#HStresstestingandexcessloadshedding). In addition, you should test whenever major architectural changes are made such as adding new dependencies or taking on new use cases. Testing frequently is the best way to know how your system scales, because the factors change as new features are added and removed.

Understanding the requirements of your system is critical to designing a scalable system:

1. **What are your Service Level Agreements (SLAs)?** Despite what many teams do at Amazon, SLAs should be well understood during design. SLAs are *not* set by first building the system and then measuring how it performs. You must understand what agreements and guarantees you will make to your clients about your behavior. For example, AWS DynamoDB makes a 99.99% availability guarantee to its clients – if they have an availability less than 99.99%, they issue a 10% credit to their clients. What would your client guarantees be? SLAs often include multiple factors, such as “return within 100ms, without error, 99% of the time.” Understanding the guarantees you will make to your clients, and the implications of those guarantees, is key to understanding how to scale your system.
2. **Do you need to scale for the peak hour of the peak minute of the peak day?** If so, why? – and is it truly critical or only a nice-to-have? Much of Amazon’s efficiency and scalability in the years since 2015 has come from asking ourselves these hard questions. We use strategies such as [Accelerated Experiences](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture/#HAcceleratedExperiences) to proactively provide a slightly-degraded experience to ensure that we scale more effectively.
3. **Can you use stale data?** Many systems can get by with slightly stale data, so [caching](https://w.amazon.com/bin/view/CEH/Scalability/#HCaching) is an important scalability consideration. Some client use cases, such as the “My Orders” page on the Retail website, cannot get by with simple caching. Customers expect to be able to place an order and then immediately see it on the “My Orders” page. To enable this, you might need to employ complex strategies like “cache busting,” or alternately not use caching at all. Understanding your customer use case is critical to understanding how to use caching.
4. **Can you do work later?** Amazon does a lot of work asynchronously from the user experience. It’s often far more resilient, reliable, scalable, and efficient to do work off-line rather than doing it while the customer is waiting. This comes with a trade-off for the user experience, so understanding the user and being customer-obsessed is critical.

There are three strategies that are generally employed when scaling, in order of preference:

1. **Avoid or defer work.** This is the ideal situation, where you build your software out of components that can be executed later, or at least defer expensive work until later. This can be done through pre-computing or caching responses to common API calls, doing shallow checks inline (such as credit card pre-auth) and then deep checks later (such as credit card settle).
2. **Horizontal** scaling is when you “scale out” by adding more components/nodes to your system as scale increases. This is the typical case at Amazon, where software scales by adding more hosts to service, more hosts to a caching fleet, or more DynamoDB capacity. This type of scaling is also best when you scale linearly or sub-linearly with a customer-facing metric (order rate, page views, and so on). If you scale more-than-linearly, then you likely should rethink the design.
3. **Vertical** scaling is when you “scale up” by adding larger resources. This is typical in single-host applications, such as some databases where scaling is generally performed by adding a larger computer with more CPU/memory.

**Warning:** The Amazon PE community does not endorse the use of vertical scaling in any business-critical application.

## Measuring scalability

“Is my system scaled for load?” is a relatively simple question to answer: test it! Tools such as [TPSGenerator](https://w.amazon.com/bin/view/TPSGenerator/) help generate artificial load and can be included as part of a deployment pipeline or run ad-hoc. This generates load to a system to help identify if and how your system reacts to having a certain amount of load. It can help you answer questions such as “what happens when my system gets 100 transactions per second? What about 1000? What about 10,000?” Running a load test periodically, especially before peak events such as the holidays, is an important part of software hygiene. More information on load and stress testing can be found in [Stress testing and excess load shedding](https://w.amazon.com/bin/view/CEH/Availability/#HStresstestingandexcessloadshedding) in the Availability section of this handbook.

All systems have hidden scalability bottlenecks. The [November 2020 Kinesis outage](https://aws.amazon.com/message/11201/) demonstrates one such hidden bottleneck: thread limitations. In this case, Kinesis front-end servers provisioned a thread for each other server, to be dedicated for communication with its peer. This is innocuous when the number of servers is small, but as the number of servers approached the operating system limitation for number of threads, servers were no longer able to communicate with each other. This then caused a cascading failure. Stress testing your system to failure is a good way to identify these hidden bottlenecks. For bottlenecks that are impractical to test for, such as provisioning tens of thousands of hosts, setting alarms on the highest or largest test that has been performed is a good compromise. For example, an alarm that went off when Kinesis provisioned more hosts than ever before would have at least alerted the team that they were beyond what they had tested. Testing, combined with techniques such as [cells](https://w.amazon.com/bin/view/CEH/Scalability/#HReducingthecostoffailure) is a way to reduce the impact of these failures.

### Breaking point

As load/traffic increases on a service it will eventually "break," even if it is scalable. Latencies for processing work will increase, while throughput will plateau and eventually start to decline. This can be shown by plotting the throughput, such as requests per second against the total latency of processing. For request/response systems, you can do this by configuring TPSGenerator to run [Find Max TPS](https://w.amazon.com/bin/view/TPSGenerator/FindMaxTps/) and then plot the latency consideration. The breaking point can be clearly seen in a graph, where it transitions from a slow linear increase in latencies to an aggressive exponential degradation. This is referred to as the "elbow."

The breaking point commonly occurs earlier than it should due to a variety of constraints that result in work queuing. Identifying and working around these constraints can considerably improve the scaling curve, shifting the elbow to progressively higher TPS before it compromises the latency. Key considerations for identifying your system's breaking point include:

* **Threading.** The threading model of the code is often the result of premature queuing and under-utilization. Are you using thread pools? How are your thread pools configured? Do you use multiple threads for a single work unit? These questions are important to consider, because if you get them wrong you can end up queuing or, even worse, deadlocking. You should ensure that you're not starting threads regularly and almost always use a ThreadPool to cache, carefully consider how your thread pool should scale, and decide what the policy should be to handle work when it can't scale any further. (Do you reject work, enqueue it in a work queue, or call thread runs?)
  + Be aware of what is going to run on the thread pool. You might need hundreds of threads for serving blocking remote calls in parallel, but for serving local computation needs you shouldn't exceed the total number of cores - 1 across  all compute-focused thread pools.  
    Avoid running input/output (especially remote service invocations) on compute-focused thread pools, because you can end up starving out the compute-centric parts of the application. This is a commonplace efficiency defect in Java since JDK8 introduced parallel streams and the ForkJoin pool. A number of JDK8 operations use the common ForkJoin pool, which is configured for local compute and can [easily lead to accidental breaks](https://dzone.com/articles/be-aware-of-forkjoinpoolcommonpool).
  + Finally, when multi-threading it's worth noting you should always check that you're actually  multi-threading. It’s common to find code that is in fact still operating sequentially, despite handing off across threading boundaries.
* **Blocking (locks).** Locks that apply cross-invocation can quickly become a bottleneck, as the more concurrency you have, the more queuing will occur for the cross-invocation locks. A prime example of this existed in the web rendering fleet in 2014, and its removal shows a [significantly better scaling curve/deferred breakpoint](https://w.amazon.com/bin/view/Users/Koturano/DPX_fleet_cost_reduction/#Promote_code_changes_to_remove_3_global_locks_in_Spring_and_Horizonte).
* **Other explicit queues.** Under heavier load a service may start to build up queues waiting to perform I/O, especially if I/O writes to local storage. Or your service may start blocking when trying to hand off to an asynchronous logger; many will only drop by default, such as Log4j2's AsynchronousAppender. You may also be calling your service through a VIP that has a surge queue configured. It is highly recommended that you disable the surge queue function from your VIP.

TPS Generator's [Reeses](https://w.amazon.com/bin/view/Reeses/) support integrates Profiler, which can help identify some of these key drivers of your system's breaking point. There are a lot of subtleties and if you're at all stuck we encourage you to reach out to your team's Senior Engineer, Principal Engineer, or if you can't find anyone, have a [principal consultation](https://w.amazon.com/bin/view/Principal/Design_Help/Consultation/).

## Traffic shaping patterns

[Systems in a service-oriented architecture (SOA)](https://aws.amazon.com/builders-library/challenges-with-distributed-systems/?did=ba_card&trk=ba_card) follow a request-response model that starts with a client on a customer device (a browser or mobile app) requesting some information and then waiting for and processing a response (such as HTML in response to a web content request or data in response to a query). This is a recursive pattern wherein invoked services subsequently become clients for downstream services, requesting more granular data needed to flesh out the complete response for the top-level client request. At scale, the resulting service call graph may fan out to hundreds of services, each of which must return responses within well-defined latency thresholds without errors. To reliably serve an ever-increasing number of requests (some Amazon systems serve millions of requests per minute under peak load), we need to engineer systems in ways that minimize the amount of work performed for each request. The extent to which we can shape the traffic to downstream services influences the extent to which the overall system can reliably handle more load while keeping costs manageable.

Traffic shaping patterns provide both availability and scalability benefits. From a scalability standpoint, the intent is to continuously operate at increasing traffic levels, with a high-fidelity customer experience, in as frugal a manner as possible. By contrast, the availability perspective focuses on gracefully adapting to anomalous conditions.

### Avoiding work

All avoidance strategies require us to make trade-offs in terms of acceptable impact to the customer experience. The extent to which we can accept these impacts depends on the availability risks we need to mitigate, the latency profile we are optimizing for, and the amount we are willing to pay to balance these concerns. The avoidance strategies below are presented in descending order in terms of their associated impact on the customer experience, and, therefore, their relative suitability for continuous or situational usage.

### Caching

Caching is based on the concept of not replicating work that was recently performed. By placing a cache in front of a service, we may be able to serve some responses from the cache and avoid re-computing the data. Effective use of caching can not only improve the application's Availability, but can also improve latency (Customer Experience) as well as significantly increase the efficiency. To learn more about Caching please refer to the [Caching](https://w.amazon.com/bin/view/CEH/Efficiency/Efficiency_and_Architecture/#HCaching) section of the efficiency wiki.

### Dynamic feature adaptation

In cases where customer-driven load begins to push services to the breaking point, it may be more desirable to serve partial responses than to outright fail requests. Consider a page with 10 features, where 3 are critical features and the rest have an ordinal ranking in descending importance. Under normal conditions, we would scale our fleets to reliably serve all 10 page features. However, under increased load or distressed conditions (such as a degraded network), we would rather serve responses containing all 3 of the critical features and as many of the lower-priority features as possible than serve nothing at all. This requires building services with sufficient feature isolation and loose coupling so as to return meaningful partial results within latency SLO tolerance.

##### More resources

* [Cards Program](https://w.amazon.com/bin/view/Card/)

#### Deferring work

Deep synchronous call stacks increase opportunities for inter-service failures and the time that originating callers must wait for responses. In some cases, you can defer completing parts of a request until after the critical path of the synchronous transaction is complete. Deferring less-critical or longer-running parts of the overall transaction is one way to scale the throughput of your system. One way to do this is by publishing an event to queue up and asynchronously perform some deferred unit of work. For example, for processing a credit card transaction, we might perform a lightweight pre-authorization in scope of the synchronous transaction and then settle the amount asynchronously.

##### Asynchronous processing

While deferring work can increase throughput as an enabler for scaling, it also requires careful consideration to ensure overall transactional integrity. Namely, we need to ensure the work will eventually complete within some specified SLO. Things to consider:

* Guarantee that the event gets delivered to a queue.
* Manage errors that occur when the deferred work executes. For example, resolve credit card settlement problems. This may require a compensating transaction to pause the purchase and initiate a workflow with the customer to resolve issues.
* Correlate logs of immediate and deferred sub-transactions for auditing and to identify opportunities for optimization.

##### More resources

* [Amazon Simple Queue Service](https://aws.amazon.com/sqs/)

### Speculative work

Within Amazon, some systems also speculatively perform work. This is generally done not for scalability, but instead to reduce latency. For example, we pre-compute a lot of the data for the Amazon detail page. This is done to reduce the customer-facing latency of rendering the page. By doing pre-compute, the work at rendering time is about formatting rather than gathering and processing information. This works well and is cost effective if we expect the data to be read more often than it is written to. Systems such as [Datapath](https://w.amazon.com/bin/view/Datapath/) can be used to precompute data.

## Horizontal scaling

Horizontal scaling is when a system “scales out” by adding more machines. This is a typical type of scaling used at Amazon, from Consumer to AWS; we scale by adding more machines in order to handle more load. This only works because we’ve designed our systems to behave this way. We also provision more compute than is strictly necessary to handle current load because we [plan for availability zone failure](https://aws.amazon.com/builders-library/static-stability-using-availability-zones/). By ensuring that we have enough machines running our systems, in distributed locations, we ensure that machine failures are handled well.

If you believe you have a situation where you can’t make your system horizontally scalable, you should reach out to a Principal for a [Principal Consultation](https://w.amazon.com/bin/view/Principal/Principal_Engineer_Design_Guidance/).

### Stateless processing

The simplest way to achieve a horizontally scalable system is to design it to be stateless. This means that your system should store no *permanent* state (logs are fine) on its disk or in its memory. Treat every single host as if it could disappear at any moment. Technologies such as [AWS Lambda](https://aws.amazon.com/lambda/) or [HEX Honeycomb](https://w.amazon.com/bin/view/Honeycomb/) make this easy: your code must be stateless by definition to use these technologies. For many systems at Amazon, this is the preferred solution; just use a technology that requires you to design your system to be stateless and it will automatically scale effectively. Typically, we try to make our business logic stateless so that the computation can be done in a trivially-horizontally scalable fashion.

Making your system horizontally scalable in this way doesn’t mean that it can scale "infinitely". Consider a system which validates that we can ship items to a specific address–it takes in an order, validates that each line item of the order can go to that address, and returns results. Because it stores no state on the host, it should scale effectively, right? Wrong: the amount of work that this system does goes up for each line item, because it has to validate each item. It scales *horizontally* with the number of orders, but it actually scales *vertically* with the size of the orders. We also need to use strategies that limit the amount of work done per request. We could limit the number of line items processed per request to put a hard limit on the amount of work done per request. This would make the system more scalable, but put more coordination work on the client. We could alternately have a separate fleet with longer timeouts and/or more memory to handle these requests. This is likely to reduce implementation effort, but increases operational load in the long term. Hard limits on input request sizes are generally preferred, as they lead to predictable behavior of the system, even if they require client coordination. A typical solution is to use pagination.

A system that has no state anywhere isn’t a common design pattern. It’s rare to have a system where all information is provided in the API and it doesn’t get data from anywhere else. Generally speaking, storage is offloaded onto AWS storage systems, typically AWS DynamoDB or Amazon S3. Using these technologies isn’t difficult, but they do take some understanding to use them effectively. AWS DynamoDB, especially, can have hot spots if not used effectively. To understand why, you need to understand [sharding](https://w.amazon.com/bin/view/CEH/Scalability/#HSharding).

### Automatic scaling

One of the advantages of a stateless system is that we can use [autoscaling](https://en.wikipedia.org/wiki/Autoscaling), which is a technique where machines are added to a system automatically as load scales up. Typically this is implemented in a "reactive" way: as resources approach some limit (for example, when CPU utilization reaches 70%), more resources are added automatically. Most AWS services such as [AWS DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html) and [AWS Lambda](https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html) have reactive autoscaling functionality. Typically, a service owner can tweak a couple of thresholds, such as the utilization threshold to add more capacity or the amount of capacity to add at a time.

While reactive autoscaling works well for services that are "leaves" in the service graph (ones without dependencies), it does not work well for the complex web of service interactions in Amazon Consumer. If each system scales reactively, it takes a while for the services deep in the service chain to scale up, as they only scale up when their clients scale, which in turn only scale when *their* clients scale. A large number of systems also scale up based upon the same factor, such as customer visits to the website, so it's relatively easy to scale everything pro-actively instead of waiting for reactive scaling. This has the advantage of reducing failures: if reactive autoscaling is incorrectly tuned it leads to failures until full scaling is complete. It also allows us to centrally manage scaling parameters. The [Scaling Planner](https://w.amazon.com/bin/view/CloudTune/ScalingPlanner/) technology manages autoscaling for Consumer; services should plan on onboarding to Scaling Planner as part of their service development.

### Startup time

Whether you're autoscaling or manually provisioned, the time it takes for a machine to be ready to serve traffic can have a major bearing on efficiency as well as how aggressively you must scale. If it takes an hour for a machine to be able to serve traffic, you must scale at least an hour ahead of load, whereas if it takes merely seconds, you can scale just-in-time. Traditional services have focused on [priming](https://w.amazon.com/bin/view/Coral/Manual/Priming/) the service before serving traffic, to improve first-request latency and latency spreads in general. This can involve downloading large datasets to operate locally, such as staging into memory for a local cache, and is often done in the name of reducing p50 request latency. Such priming can significantly increase the time it takes to add a new machine to a system. Key startup time considerations include:

* **Time to First Request (TTFR):** How long does it take to add a new instance? If it takes too long, it forces you to rely on predictive or planned autoscaling rather than reactive autoscaling.
* **Time to provision:** How long does it take to provision your environment? This is generally a function of three components:
  + **How much of the stack you have to provision.** Provisioning the stack from the operating system, as with EC2, will take much longer than provisioning it from the runtime, as with Lambda.
  + **How big your resources are.** A 50 MB function is 1/100th the size of a 5GB one, and can be provisioned much faster. Ensuring that your deployment image contains only what you need is critical. For example, only deploying the AWS SDKs for the services you use, rather than the entire AWS SDK, can save over 100MB. How many pieces you need provisioned.**1 50MB resource will be much faster to provision than 500 100KB ones.**
* **Time to start your process:** How long does it take for your process until it can start to receive traffic? There are three main considerations here:
  + **Choice of language/runtime.** Some languages, such as Java, are slower to initialize than others.
  + **What gets initialized and how.** Using frameworks that do a lot of dynamic discovery on startup, such as Spring XML, can significantly extend the process initialization time.
  + **Priming before serving traffic.** Initialization functions that do priming actions such as downloading and caching configuration can drastically increase starting time.

To address these concerns in general, you should look to see what you can move into your build process:

* Choose a higher order runtime to cut out the need to provision all the resources that the managed runtime has waiting for you.
* Prune or minimize resources manually (for example, taking dependency on specific AWS Clients you need rather than the full SDK) or automatically via a build tool (such as [minification](https://en.wikipedia.org/wiki/Minification_(programming))). You'll have less to deploy.
* Build a single image rather than having a manifest based retrieval from a package manager. You'll be able to provision it faster.
* Keep your initialization code simple and direct to start faster. Likewise, consider choosing a language/runtime that's designed to start quickly.
* Keep your code stateless with configuration and cached resources kept remote to the process. You'll be much better positioned to start quickly.

### Sharding

It’s important to understand the concept of sharding, even if many teams are only going to *use* a sharded datastore, rather than *implement* sharding. Sharding is a practice of dividing data and/or work amongst nodes/hosts using a stable algorithm. This works well when the data being sharded is keyed by something with high cardinality, such as customer identifiers. By assigning each customer a shard, we can put many customers on a single shard but easily remember which shard we put them on to go find data later. This lets us be efficient by storing data for many customers on the same machine, but lets us scale up by just adding more machines as we add more customers. Each shard in this methodology has a piece of the entire dataset. No one machine knows everything, but since we rarely, if ever, access the full dataset, it doesn’t matter that it’s not all in one place. Each individual shard has a specified SLA and/or limitations; for example, AWS Kinesis shards are limited to 1000 writes per second. You scale by adding many shards. Because each shard has well-defined SLAs, it’s easy to estimate how many you need and how they might fail. Scaling is also easy, as it is merely adding more shards. By enforcing limitations on the scale of their shards, Kinesis ensures that each shard behaves within its bounds.

In AWS DynamoDB, the most commonly used sharded datastore at Amazon, data is sharded by the “partition key.” Scaling is done automatically, and users specify how much capacity they need, rather than how many nodes they want DynamoDB to provision. DynamoDB automatically provisions the right number of nodes given the capacity the user requests. Picking the right partition key can be tricky, and AWS has [some tips](https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/) about doing so. Shards in DynamoDB are also replicated, so that users don’t have to worry about what happens if one of the machines that service the shard fails. The system gracefully handles it for you.

### Reducing the cost of failure

[Cell-based architecture](https://w.amazon.com/bin/view/CellBasedArchitecture/) is a technique first developed within AWS to reduce the size of failures for their services. Because the design had many customers sharing a single service instance, any outage impacted every single customer. Cell-based architecture is a type of sharding where the service limits the number of use cases on a single "cell" to a known quantity, and then scaling is performed by adding isolated cells. For example, instead of having a single instance of the service running in us-east-1, there are many equally-sized instances (the cells). Resources (like customers) are assigned to a single cell, so if a cell fails it only impacts a fraction of customers. This is demonstrated in [COE 89069](https://www.coe.a2z.com/coe/89068/content) where a single cell of Cloud Directory failed; this caused a full outage for 1/6 of their customers while the remaining 5/6 of customers were completely unaffected.  There are a number of [benefits](https://w.amazon.com/bin/view/CellBasedArchitecture/#Benefits) and [implementation considerations](https://w.amazon.com/bin/view/CellBasedArchitecture/#Implementation) that teams should familiarize themselves with if they're considering a cell-based architecture.

[Shuffle sharding](https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/) is a form of sharding/cellularization where workloads are sharded as well as isolated in order to provide tenant isolation and avoid the [noisy neighbor problem](https://en.wikipedia.org/wiki/Cloud_computing_issues#Performance_interference_and_noisy_neighbors). While this is an interesting algorithm developed by AWS, most engineers just need to be aware of the concepts and not actually implement shuffle sharding.

Cell-based architecture and shuffle-sharding are techniques with a lot of subtleties, and teams considering using them should have a brief [Principal Consultation](https://w.amazon.com/bin/view/Principal/Principal_Engineer_Design_Guidance/) to ensure that the use-case is well understood.

## Vertical scaling

Vertical scaling is when you get a bigger machine to “scale-up” your system when it needs to grow. Vertical scaling has a single point of failure by definition. It can work well when the function performed by your system is not business critical, as we don’t want a single point of failure in a business-critical system. Because of the single point of failure, any system that scales with vertical scaling should expect to be available at most 99% of the time. Understanding your SLAs and requirements are key to understanding whether this limitation is allowable. It is not acceptable for any customer-facing system within Amazon, and thus is at best only applicable to back-end processing systems such as those that perform non-critical work and consume from a queue.

Traditionally, Amazon did this type of scaling, through buying bigger Oracle database hosts for storage. After the [Rolling Stone project](https://w.amazon.com/bin/view/RollingStone/) was completed in 2018, Consumer migrated all business-critical functions off of these solutions. Vertical scaling relies on the fact that, for some business applications, the growth of the business is slower than [Moore’s Law](https://en.wikipedia.org/wiki/Moore%27s_law). While this may be true in some situations, as the business evolves it can cease to be true. A common scenario in a vertical scaling world is that you run out of growth options. For example, in pre-Rolling Stone Amazon it was typical for a team to request bigger and bigger database hardware as their system scaled. But what do you do if you’re on the biggest hardware and can’t grow? You’re completely out of luck and need to do something in an emergency. This is not a good situation to be in.

**Important:** Vertical scaling is discouraged at Amazon, and we ask that any system that plans to scale vertically [get a principal review](https://w.amazon.com/index.php/Principal/Principal_Engineer_Design_Guidance).