Executive Summary

Whether it's an electronic medical record (EMR) system or a smart pill dispenser, failures in software-based medical solutions can delay or disrupt patient care and even jeopardize their well-being.

For that reason, architects of the software used in medical systems or devices need to take a page from the Netflix and Amazons of the world, companies that have pioneered innovative methods to ensure the software that runs critical systems is both resilient and reliable.

Resilience is a system’s ability to keep functioning as needed despite a change such as a sudden spike in users or the failure of a database. It also describes a system's capacity (usually expressed in percentage terms) to function properly under a given set of conditions for a given period.

In this white paper we describe the need for resilience and reliability in software-based healthcare solutions, offer eight best practices for achieving it and provide design recommendations for resilient and reliable systems architectures.
Resiliency and reliability in healthcare

With the advent of digital technologies, the abilities of software to run nonstop and to allow users to engage with it anytime, and from any device, are essential. In addition, the nature and scope of software functions have expanded considerably, adding to the need for resiliency and reliability. Among the changes driving this need are:

- **The distributed nature of software**, including service-enabled, service-oriented and microservices architectures. But the more distributed the software, the more potential points of failure, increasing the need for higher levels of fault tolerance.

- **More heterogeneous technology landscapes** comprising solutions from more vendors, increasing the number of interfaces and potential problems.

- **More complex functions**. These include tracking a patient's condition via a wearable device, using artificial intelligence (AI) to diagnose a condition or recommend an action based on that information, or waiting for prior authorization status when the patient is at an in-patient facility. Determination of prior authorization status can take more than a week when multiple providers and payers are involved in a patient’s care.

- **The use of cloud and cloud-related technologies** that facilitate auto-scaling, failure detection and auto correction.

Because these more distributed, complex systems have more potential points of failure, and less predictable failure modes, resilient and reliable software architectures must accommodate the inevitable failures by being fault-aware, fault-tolerant and self-healing. Among the healthcare systems enabled by these technological advances are:

- **EMRs and electronic health records** which must be highly available to give caregivers real-time access to the information they need to serve patients. These include the patient’s medical history, diagnoses, allergies, medication history, lab results and care plans. Such systems often include components that provide administrative, clinical and financial integration with EMR functions.

- **“Smart” dispensers** that tell patients when to take a medication and monitor whether they are taking them properly. Because of the threat to patients’ health should these dispensers malfunction, they and the software within them must be able to keep operating despite any conceivable failure.

- **Clinical decision support systems** that guide caregivers in choosing the appropriate treatment for a given condition, whether based on existing rules or the use of AI and machine learning. This could include clinical-effectiveness-based guidance to the caregivers for diagnosis, prescriptions, lab orders and procedure administration.

Because these more distributed, complex systems have more potential points of failure, and less predictable failure modes, resilient and reliable software architectures must accommodate the inevitable failures by being fault-aware, fault-tolerant and self-healing.
Best practices for resilience and reliability

Eight best practices, or architecture design patterns, enable the graceful degradation of systems and their ability to self-heal after failures. They also enhance system stability and scalability, decouple software modules, make the system more distributed, and enable more efficient data storage and utilization.

1. **Isolation**: This is the practice of building new features or functions behind switches or “feature flags” that allow a function to be disabled without changing the code, thereby isolating a feature or function that has degraded. This prevents the cascading failures that often cause the most catastrophic system outages with the highest risk to patient care. Combining such feature flags with the division of a monolithic application into microservices can also significantly isolate failures.

2. **Modularity**: Making every module of software responsible for the smallest possible subset of functions, and minimizing modular interdependence, further reduces the damage from any module failures. It also makes it easier to maintain and enhance each module with a minimum of code changes.

3. **Redundancy**: Redundant databases, application servers and other essential application components provide alternate execution paths in the event of the failure of any part of the system.

4. **Repeatability**: Resilient systems are built on consistent, well-understood and repeatable processes and policies. These range from continuous integration, deployment and testing pipelines to applying infrastructure-as-a-code to immutable infrastructure. Repeatability helps minimize errors, reduce recovery time and provide a record of a system’s previous state and how it failed, which speeds troubleshooting and recovery.

5. **Health checks**: Periodic health checks gather both local and dependency information to determine if a component can successfully handle a service request. They typically need intelligence to strike a balance between the failure detection and reaction to it so that the transient errors do not end up isolating that component. Health checks should provide trend tracking to find service degradation early and apply corrective actions.

6. **Caching (statelessness)**: Caching of a system’s state (information about the status of its components and transactions) reduces the resources the system consumes. This improves scalability, reduces loads on downstream systems, helps isolate components and slows degradation after a failure because the information is readily available within the cache. Even if the server/system components are not fully functional, the cache will help the application to run – although with limited information availability. In-line caches with an eventual consistency model also help make the client stateless, which contributes to creating a decoupled system.

7. **Time-outs, retries and back-offs**: Time-outs terminate the connections that degrade performance rather than allowing those failed connections to halt an entire application. Retries attempt to reprocess the terminated connections in the hope that the problem has resolved itself, while back-offs are when algorithms gradually decrease the frequency of retries to avoid network congestion and the freezing of an entire application. Together, these patterns ensure that transaction failure is minimized, degradation is not aggravated and time to recover is reduced.

8. **Input vetting**: Evaluation of all inputs to ensure they are not harmful or malicious goes a long way toward protecting systems against attacks and ensuring a stable and secure platform.
A framework for resiliency and reliability

A multidimensional framework can help focus software development processes for maximum resilience and reliability, taking into account structural, behavioral and operational considerations (see reference architecture diagram in Figure 1), which we’ll now examine separately.

<table>
<thead>
<tr>
<th>Isolation</th>
<th>Modularity</th>
<th>Redundancy</th>
<th>Regional affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blast radius config</td>
<td>Retry - circuit breaker</td>
<td>Restructure</td>
<td>Reinitialization</td>
</tr>
<tr>
<td>Proxy management</td>
<td>Load management</td>
<td>Decoupled services (MSA)</td>
<td>Queue</td>
</tr>
<tr>
<td>Monitoring, alerting, notification, dashboards</td>
<td>Prediction, self-healing, graceful degradation</td>
<td>Automated health, checks, measurements, threshold management</td>
<td></td>
</tr>
</tbody>
</table>

Source: Cognizant
Figure 1
Structural

The structural aspects involve the following:

- Decoupling the compute and data components enables the platform to avoid cascading effects of a failure. Getting the functional design right is often a challenge, because if there are too many dependencies among application components, additional resilience measures will fail to create a more robust system. Guard against building too much reusability into an application. While it is very desirable within a process boundary, it also may create a strong coupling that exhibits undesirable properties across process boundaries.

- Isolating compute components helps prevent problematic components from causing wider failures, decouples components depending on the load and helps scale the system as needed. Isolation facilitates modularity and decoupling, which in turn helps in graceful degradation in adverse scenarios.

- Redundancy of key components, such as database or compute servers, enables the dynamic routing of requests depending on the load and helps scale the system as needed. Managing redundancy across regions, such as zones or network clusters, also increases availability. Data caching and the use of content delivery networks reduce compute, database, and network loads, and allow services to provide partial data (mandated for the operation or function) if the core service providers are not available.

Behavioral

This aspect refers to how system components react to issues that threaten application availability. They include:

- Determining the “blast radius” of compute or data components made unavailable by various failures.

- Using “circuit breakers” that automatically disconnect components to avoid cascading failures.

- Fault management, reconfiguration, checkpoint recovery, state diversity, design diversity and microservices.

- Agents that schedule last recovery points for graceful degradation.

- Stateless services or applications decoupled from the underlying infrastructure or environment.

Operational

These aspects, which refer to processes used after the system has been deployed, include:

- Detecting failures and eventually predicting failures and automatically isolating the failures, recovering from them and repairing the failed components.

- When using the monitoring index for the hardware and software components, vitals are kept under the radar. For instance, if the memory used by a component or an instance is not recycled, the application may run out of memory needed to execute and service further requests.

- Addressing failures that interrupt business processes. For example, if the business rule responsible for checking “member eligibility” throws an error, the monitoring agent can alert the system administrator or an auto-agent that can remediate. Timely action can then be taken either to switch the invocation to another instance or to bring the faulty process down with an appropriate user-friendly message to the consumer without the failure’s effects cascading further.

- When addressing operational aspects in highly distributed systems, think beyond the system either operating normally or failing. Instead, focus on how to build a system that can tolerate inevitable and continual failures, and recover transparently. AI algorithms that predict failures can trigger roll-back or roll-forward scripts to either prevent the fault or speed recovery.

- In addition, the software itself can be made fault tolerant by the use of methods such as recovery blocks, N-version programming and self-checking software.
## Failure triggers and design solutions

<table>
<thead>
<tr>
<th>Triggers that can reduce reliability and resilience</th>
<th>Design elements that prevent or repair failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial or total system outages.</td>
<td>Structural factors such as redundancy and fail-over to other regions, distributed cache, retry-circuit breakers.</td>
</tr>
<tr>
<td>Compute/network/storage capacity not scaling to meet demand.</td>
<td>Load balancing, traffic management, queuing, stateless design.</td>
</tr>
<tr>
<td>Lost user sessions, frequent data retrievals slow response time.</td>
<td>Distributed cache, content delivery networks, stateless design and caching.</td>
</tr>
<tr>
<td>Long wait time for threads within applications leads to bottlenecks, loss of in-flight transactions.</td>
<td>Queues, agents/schedulers for asynchronous communication.</td>
</tr>
<tr>
<td>Single points of failure.</td>
<td>Modularity, isolation, decoupled services, automated health checks.</td>
</tr>
<tr>
<td>Long recovery times, data loss following system failures.</td>
<td>Distributed caches, version and checkpoint provisions, repeatability, roll backward vs. roll forward, active-passive clusters.</td>
</tr>
<tr>
<td>Unnecessary demands on the services layer.</td>
<td>Design state management with techniques such as caching to reduce the number of avoidable hits.</td>
</tr>
<tr>
<td>Reactive approach to identifying system issues leads to avoidable downtime</td>
<td>Leveraging operational capabilities such as logging, tracing, auditing, monitoring, visualizing and alerting to proactively eliminate avoidable downtime.</td>
</tr>
<tr>
<td>Inappropriate selection of integration patterns without considering the business context and nonfunctional requirements.</td>
<td>Use of behavioral patterns such as asynchronous queues/topics rather than real-time service integration, where appropriate, to decouple components and improve resiliency.</td>
</tr>
</tbody>
</table>

Figure 2
Measuring Resilience and Reliability

As with any software, platforms engineered for resiliency and reliability will change over time. Ensuring they continue to meet business needs requires measurement.

While the required metrics are evolving, one important indicator is mean time between failures (MTBF). This in turn consists of mean time to failure (MTTF), which is the time period between two consecutive failures, and mean time to repair (MTTR), which is the time required to fix the failure. As failures can never be entirely prevented and the goal is to recover quickly from them, success consists of reducing mean time to recover rather than increasing MTBF.

Implementing such measurement requires:

1. **Determining resilience thresholds.** Which transactions or services are most critical to sustain for users? How quickly can the disrupted services recover? What is the proportion of requests that can be lost during a disruptive incident?

2. **Static analysis of resilience capabilities and flaws.** Determine which resilience capabilities, such as isolation and redundancy, exist within each component, and correlate those resilience elements with the resilience requirements for that component. This can be achieved, for example, by validating whether infrastructure routing procedures are in place to route service requests to the most robust components.

3. **Dynamic analysis of system behavior under disruptive conditions.** This involves simulating a set of disruptive conditions the system is designed to withstand, evaluating its behavior under those conditions and correlating that behavior to the resilience thresholds.
A reference architecture for resiliency and reliability

All these patterns and practices should be applied comprehensively to build a resilient and reliable software application platform. Its main elements will include:

- Managing and distributing the load using a load balancer (LB) that can not only manage the routing of traffic but can be configured to handle session management. A properly configured load balancer can also break a circuit if it finds one of the outbound end points is not responding.

- Use of a proxy server that, depending on usage levels, can double the security constructs through DMZ firewall rules and by adding a proxy server, using port redirection logic, rate limiting and origin/protocol filters.

- Application programming interfaces (APIs) and orchestration engines to provide on-demand, asynchronous communication through handshakes. Event engines and queuing platforms also support resiliency and reliability.

- Regional replication allows system components in another geography to automatically take over for a failed component. While this may cause latency, it at least keeps critical services running to prevent business disruption.

- Use of a content directory network or caching provides multiple copies of information to manage the load on active servers and provides data to users or applications if the information services have failed. This is most useful when the use of “stale” data is acceptable.

- Containerization using tools such as Docker or Kubernetes helps isolate, decouple and independently scale compute components.

- Implementing stateless and self-healing components that do not hold session information. This breaks the compute into smaller chunks, enabling distributed computing and eliminating disruption due to contextual user sessions. Once the business processing is decoupled, using active monitoring features across the services enables the system to recover automatically (i.e., to be self-healing).
Insisting on failure analysis and recovery mechanisms, and understanding the consequences of a failure, are essential to meeting business needs long-term.

Figure 3 illustrates an end-to-end mapping of key components that make up an enterprise-wide distributed system built for resilience and reliability. Many organizations will still rely on legacy infrastructures that were not optimally architected. They can improve this existing infrastructure through these methods:

- Deploying logically separated components onto isolated physical infrastructure to enable greater scalability.
- Use of single sign-on to host a set of related applications that can be maintained on an independent computer/network/storage infrastructure. This provides a single entry point for users while minimizing dependency among the applications.
- Injecting audit controls for critical workloads, analyzing the logs and acting on them (such as with alerts or rule-based corrective actions) rather than reacting to problems.
- Implementing infrastructure-as-code to automate and speed infrastructure provisioning and management using automated monitoring and scaling technologies.

When deploying resilient and reliable software, build in time for failure analytics, mitigation and testing. Most software development and evolution happens in a time-constrained environment, with stakeholders favoring features that work over features that last. Insisting on failure analysis and recovery mechanisms, and understanding the consequences of a failure, are essential to meeting business needs long-term.

**Dynamic resiliency pattern for a distributed system**

![Dynamic resiliency pattern for a distributed system](image)

Figure 3
Fault vulnerability analysis

Our proprietary Fault Vulnerability Analysis framework provides a structured way of proactively identifying and addressing failures and faults in applications and infrastructure. Key steps include:

- Identifying potential failure modes in the overall application architecture based on key failure and impact dimensions, such as sudden spikes in load or volume, latency between dependent components, data inconsistencies and error conditions.
- Attending to existing controls built into the application and infrastructure to proactively detect failures.
- Studying existing recovery procedures and remediation controls that will help rapid recovery from failures.
- Understanding gaps in fault tolerance, prevention, detection and recovery.
- Identifying solutions and improvement opportunities.
- Understanding the impact of the identified failures on end users and the business, along with the criticality of the impact.
- Studying existing recovery procedures and remediation controls that will help rapid recovery from failures.

Healthy systems, healthy patients

Between the rise of digital technologies and COVID-19 restrictions on personal contact, more healthcare delivery and its supporting applications are moving online. Because of these systems’ potential impact on health and well-being, those who develop and maintain them must pay even greater attention to their resiliency and reliability.

Because distributed systems are, by nature, complex and unpredictable, application architects must design features such as isolation, modularity, circuit breakers and caching to reduce the likelihood of failure, as well as monitoring and automatic retry mechanisms to reduce the impact of the inevitable failures. They must not only build these features into every layer of the application architecture, but throughout the software lifecycle, revisiting these needs as business needs and technology change.

To learn more, read our white paper on Resilience Engineering as an IT Cultural Discipline.
Best Practices for Resilient, Reliable Software Engineering

- Include resilience and reliability in every phase of the software development lifecycle, from design to deployment to monitoring.
- Build resilience and reliability into every architectural layer including the infrastructure layer (redundant storage), data layer (clustered databases), services layer (load balancing, auto-scaling) and integration layer (stateless sessions and clustered sessions).
- Incorporate design features such as “circuit breakers” and “bulkheads” to quickly identify and isolate failed components.
- Use of stateless design.
- Timeouts to reduce thread waits.
- Adopt failure-tree analysis methods such as failure-mode effect analysis, fault-tree analysis, event-tree analysis, failure-mode effects and criticality analysis.
- Test for resilience and reliability as the software evolves.
- Consider low-level programming best practices, such as:
  - Exception or failure handling modules designed to watch for and remediate known failures.
  - Optimal query process swarming is key for resources management. Swarming of processes or threads with optimal utilization of heaps helps manage compute and memory. Such improved resource management helps reduce the potential for failure.
  - Performing periodic volume-stress testing to understand performance degradation thresholds.
  - Instrument applications to monitor trends, such as connection pools and memory, and CPU usage to predict failures.
  - Asynchronous design using queues, messaging, batches and event-driven programming to avoid single points of failure and allow graceful system degradation.
References


* Resilient principles: [https://www.resilientdesign.org/the-resilient-design-principles/](https://www.resilientdesign.org/the-resilient-design-principles/).

* Three key patterns of software resilience: [https://medium.com/@buildiumlife/3-key-patterns-of-software-resilience-ae01f191a29e](https://medium.com/@buildiumlife/3-key-patterns-of-software-resilience-ae01f191a29e).

* Toward successful resilient software design: [www.infoq.com/articles/towards-resilient-software-design](http://www.infoq.com/articles/towards-resilient-software-design).

About the authors

Amber Marfatia
Enterprise Architect, Cognizant Healthcare

Amber Marfatia is an Enterprise Architect within Cognizant Healthcare, working on product engineering for the U.S. healthcare market. He has worked for over 16 years in global technology consulting and architecture in the healthcare, insurance and banking industries, focusing on multicloud architecture, product development and systems integration. Some areas of interest and activity for him are in-depth design and development in the AI/ML and conversational AI spaces and enterprise-wide digital modernization. Amber has a bachelor’s degree in physics and a master’s in computer science from South Gujarat University, Surat. He also has TOGAF 9.1 and Azure specialist certifications. Amber can be reached at amber.marfatia@cognizant.com | https://www.linkedin.com/in/ambermarfatia.

Rajnarayanan C K
Chief Architect, Cognizant Healthcare

Rajnarayanan C K is the Chief Enterprise Architect within Cognizant’s Healthcare Practice. He has over 20 years of architecture and consulting experience in software and product development for large healthcare payers across the U.S. Raj is truly passionate about software engineering and nonfunctional requirements, with specialties including cloud development with platform as a service (PaaS), microservices architecture, legacy integration, data exploration, analysis, machine learning and SDLC automation. Raj has a BE in electrical engineering from PSG College of Technology and an MTech from IIT, Madras, and has Oracle Master JEE Architecture and AHM 250 certifications. He can be reached at Rajnarayanan.Kannan@cognizant.com | https://www.linkedin.com/in/rajnarayanan-c-k-29662714/.
About Cognizant Healthcare

Cognizant’s Healthcare business unit works with healthcare organizations to provide collaborative, innovative solutions that address the industry’s most pressing IT and business challenges — from rethinking new business models to optimizing operations and enabling technology innovation. As a global leader in healthcare, our industry-specific services and solutions support leading payers, providers and pharmacy benefit managers worldwide. For more information, visit www.cognizant.com/healthcare.

About Cognizant

Cognizant (Nasdaq-100: CTSH) is one of the world's leading professional services companies, transforming clients’ business, operating and technology models for the digital era. Our unique industry-based, consultative approach helps clients envision, build and run more innovative and efficient businesses. Headquartered in the U.S., Cognizant is ranked 194 on the Fortune 500 and is consistently listed among the most admired companies in the world. Learn how Cognizant helps clients lead with digital at www.cognizant.com or follow us @Cognizant.