Redbooks
ibm.com/redbooks

Accelerate z/TPF Application Modernization with Hybrid Cloud

Mark Gambino

Luanne Jagich

Jamie Farmer

Kelly Paesano

Connie Walberg

Josh Wisniewski

Cloud

IBM Z

IBM

Point-of-View

Stepping up your digital transformation, business agility, and productivity

Highlights

- **Better together:** Modernize IBM z/Transaction Processing Facility (z/TPF) applications faster at lower cost and less risk by combining the strengths of the IBM Z® platform and cloud.
- **Proven best practices:** Learn prescriptive approaches to accelerate your application modernization journey.
- **Reference architectures:** Study and deploy the solution that is right for your enterprise across z/TPF, the IBM Z platform, and cloud.

What is the best strategy to achieve an effective digital transformation of your business? How do you make your business more agile, productive, and less siloed so that you can provide engaging customer experiences rapidly and at lower costs?

> **Answer:** *Accelerate your IBM z/Transaction Processing Facility (z/TPF) application modernization and leverage hybrid cloud.*

This IBM Redbooks® publication provides an overview of the IBM® strategy to help you modernize z/TPF applications faster at lower cost and at less risk by using IBM Z and public cloud solutions together in your modernization journey. This publication includes prescriptive approaches, best practices with supporting technologies, and reference architectures to help businesses achieve the right outcome.

Businesses need rapid digital transformation

According to a 2021 IBM Institute for Business Value study, “4 of 5 executives say their organizations need to rapidly transform to keep up with competition, which includes modernizing mainframe-based apps and adopting a more open approach.”[1] To achieve this critical business transformation with z/TPF and cloud, organizations must be able to do the following tasks:

- Accelerate time-to-value with a progressive approach to modernization.
- Increase productivity and agility that is centered on a DevOps culture.
- Address skills gaps by leveraging z/TPF and IBM Z support for programming languages, common continuous integration and continuous delivery (CI/CD) toolchains and development practices, and standard approaches to IT automation.

IBM can help you meet these goals by providing the tools and methodologies to help you transform your business by embracing hybrid cloud and leveraging a continuous application modernization journey.

[1] https://www.ibm.com/downloads/cas/7BJPNGND

The z/TPF application modernization journey begins with the same entry points as the overall IBM Z platform, which are described in the IBM Redbooks publication *Accelerate Mainframe Application Modernization with Hybrid Cloud*, REDP-5705. Dozens of z/TPF modernization patterns have been developed to address these goals. Before we describe tools and methodologies, we begin with the differentiating value of centering your modernization journey on hybrid cloud and the IBM Z platform. We also highlight the advantages of z/TPF architecture that make it the “best-fit” for high-volume and secure transaction processing workloads to help you understand the “art of the possible” for z/TPF application modernization.

Embracing hybrid cloud

Hybrid cloud has become the leading architecture for enterprises because it enables greater productivity, faster insights, better decision-making, and enhanced employee and customer experiences. z/TPF with other IBM products and services are designed to maximize the business value of hybrid cloud and AI for clients. Overall, when the IBM Z platform is integrated into a hybrid cloud approach, it can deliver more than 2.5 times the value of a public-cloud alone.

You can maximize the benefits of a hybrid cloud approach with these application modernization initiatives:

- Simplify access from digital channels.

  Simplify access to z/TPF applications and data from cloud services to support new digital channels that can enhance customer experiences.
- Share real-time information between z/TPF applications and the cloud.

  Providing customized offers that can attract customers and differentiate your business is a key objective for most businesses. Increasingly, this goal requires real-time exchange of information between core business applications running on z/TPF and digital front-end applications running on cloud.
- Leverage DevOps for z/TPF applications.

  Embrace an enterprise DevOps model that includes developing and modernizing z/TPF applications to increase speed and agility for greater developer productivity.
- Automate IT across the IBM Z platform and cloud.

  IT automation has played a critical role in large-scale enterprises for decades, and it is now essential because businesses have adopted a hybrid cloud architecture to support their digital initiatives.

Best-fit approach to application modernization

The ideal approach to z/TPF application modernization leverages the best of the IBM Z platform and the innovation of cloud.

Through decades of innovation across a vertically integrated stack, z/TPF running mission-critical transactional workloads on the IBM Z platform demonstrates high throughput, availability, and security for the following reasons:

- Data privacy, security, and compliance.

  Data can be encrypted in all contexts: in flight, at rest, and in use. Data is protected from being compromised, either accidentally or deliberately, from inside and out. Demonstrating z/TPF compliance is simplified with automated evidence-gathering and reporting capabilities.
- Transaction processing requires near-zero downtime.

  The z/TPF system is designed for resiliency. Because of the z/TPF architecture, workload changes can be handled seamlessly and application changes can be deployed dynamically without an outage.

- Transactional consistency and integrity.

  The z/TPF system and database architecture can process up to 16 billion encrypted IBM z/TPF transactions and update 150 billion database records per day in a secure and strict consistent manner.

- Scale and performance.

  The IBM Z platform is a vertically integrated hardware and software stack that is optimized for performance and efficient scalability. z/TPF supports low latency transactions at a massive scale.

- Cost optimization.

  Respond faster to customer demand and keep costs down while leveraging the efficiency of the IBM Z platform as you scale. Cost-effective solutions, including z/TPF Dynamic CPU support, can elastically scale and deliver capacity on demand when workload spikes. Specialty priced general-purpose engines, which are known as z/TPF Transformation Engines (TEs), can be used for modernization at a reduced cost.

- AI for real-time insights at scale.

  Leveraging z/TPF and the IBM Z platform can help to increase business value by bringing AI into your transactions, for example, to help detect fraud before a transaction completes.

Advantages of the z/TPF architecture

z/TPF provides differentiating value to an integrated, hybrid cloud strategy. Understanding the fundamental differences between z/TPF and other platforms is critical to achieving the best business outcomes in a “fit-for-purpose” approach to application modernization.

The z/TPF architecture is designed to minimize transaction latency and cost per transaction while maximizing resiliency, security, and the ability to grow your business workloads. Key aspects of this architecture include the process execution model, database architecture, memory model (including the ability to leverage very large memory), networking and middleware models, and security model. These fundamental architectural differences between z/TPF and other platforms enable z/TPF to scale up to handle hundreds of thousands of transactions per second while still maintaining the low and consistent response times that are required to meet service-level agreements (SLAs).

The choice of process execution model matters

On most platforms, as shown Figure 1, there are separate, long-running, and multi-threaded processes, and each one handles different functions. When one process must call a function running in another process, an inter-process communication (IPC) method is used, which requires input and output data to be copied from one process to another one. In this model, latency is incurred from scheduling the called process thread and the waiting process thread on return.

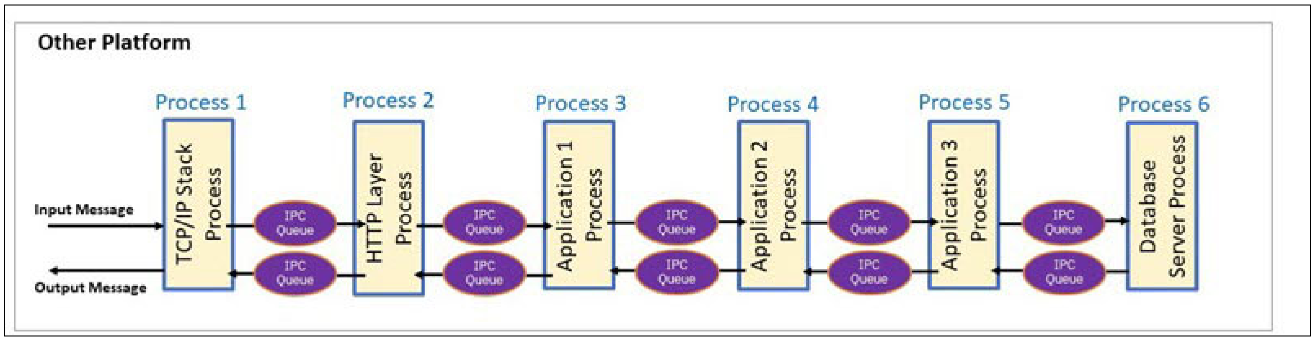


Figure 1 Process execution: Other platforms

In z/TPF, as shown in Figure 2, each typical transaction runs in its own single-threaded process and all application and system components that are involved with processing that transaction share application memory. Therefore, there is no added latency or loss of control when one application program calls a system service or another application program. Because all the components that process the transaction have access to the same memory, the overhead of copying input and output data is avoided.

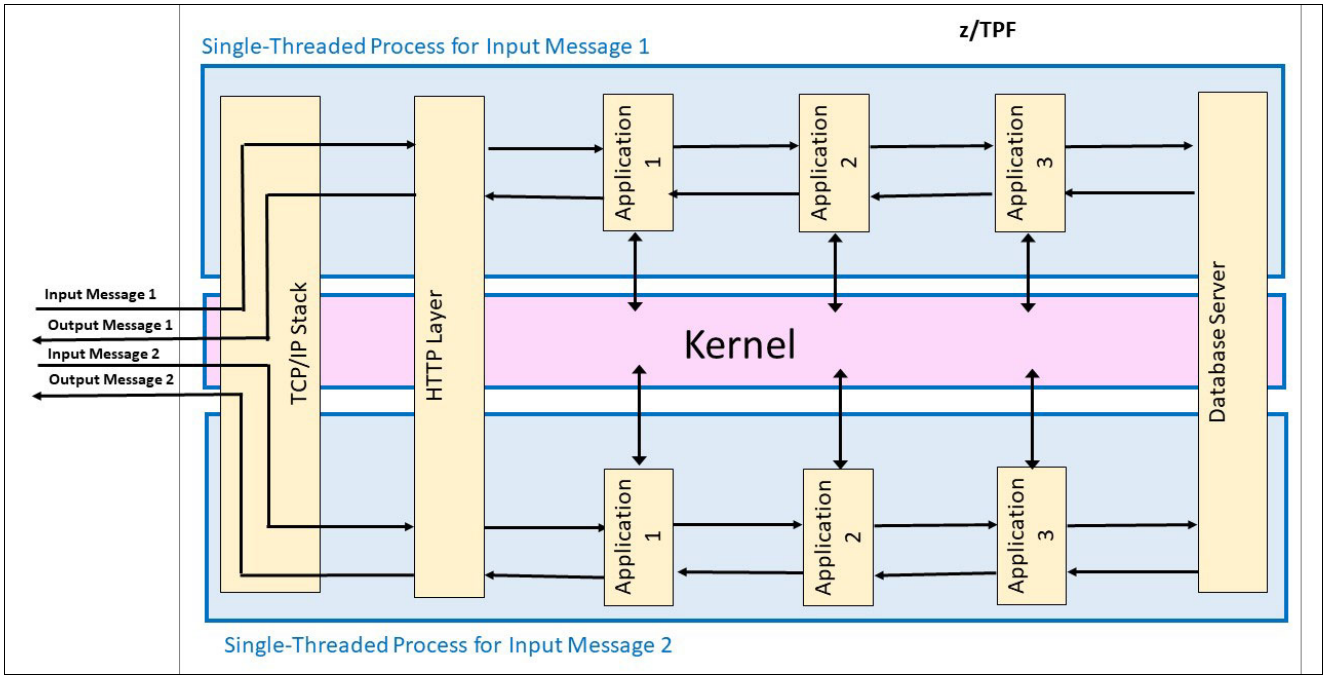


Figure 2 Process execution: z/TPF

There are many advantages to the z/TPF process execution model, such as the following ones:

- Minimized transaction latency: A transaction can go through dozens of programs without giving up control.
- Reduced CPU consumption: There are no IPC mechanisms that are involved, and input and output data does not need to be copied across processes.
- Reduced need for installed CPU capacity: Multiple workloads can be handled concurrently, which maximizes CPU resources. On other platforms, workloads require dedicated servers, which might be underutilized.
- Better security: True transaction isolation is achieved because each transaction runs in its own process. A transaction’s data cannot be seen by other transactions, and the transaction cannot see another transaction’s data.
- More resilient: When a transaction fails because of an application error, only that transaction is impacted. Other transactions continue to be processed without requiring external monitors to restart application or system processes. In addition, unexpected workload spikes are handled automatically by shifting resources where they are needed.
- No downtime: New versions of applications can be dynamically loaded without any system or application outages (no need to stop and restart the application). You have continuous deployment of new business functions as often as you want.

For more information and examples, see “The choice of process execution model matters” on page 63.

The z/TPF database architecture

Another advantage for choosing z/TPF is its database capabilities, which are secure, centralized, highly available, and able to update millions of database records per second in a strict consistent manner. Whether your workload is financial systems processing trillions of dollars in transactions per year or other services people rely on every day, z/TPF is designed to manage that data with accuracy, performance, and scale. In summary, z/TPF supports high volumes of write-intensive workloads that must update multiple databases in a highly available environment (including disaster recovery (DR)).

There are many advantages of the z/TPF database architecture, including the following ones:

- High availability: Two copies of data are kept on separate disks so that a copy of all data is accessible if a disk or entire disk control unit fails.
- Centralized: There is no partitioning of data. Applications can update multiple logical databases (for example, flight inventory and flight reservation record databases) in a strict consistent manner without the overhead and latency that is associated with processes like 2-phase commit.
- Strict consistent data: Applications always read the latest copy of data. With other database products that use an eventually consistent model, you must manage the risk that applications might read old data and you must also develop code to handle data conflicts.
- Proven ability to scale: z/TPF minimizes the time that a database lock is held, making it possible to process more transactions per second on z/TPF than other platforms. For many years, z/TPF systems have performed millions of database read and update operations per second to disk, plus millions more database read operations per second that can be satisfied from the Virtual File Access (VFA) memory cache in z/TPF, which avoids the need for I/O.
- Faster access to data: z/TPF applications and databases are colocated in the server, unlike a distributed architecture where there are separate application servers and database servers that require network flows to communicate with each other.
- Reduced CPU consumption: For a z/TPF application to access the z/TPF database, only a local call is needed. In a distributed environment, the database client layer in the application server must make an external call to the database server. The overhead of the external call is increased further if it must be done over a secure connection.
- Secure: Entire databases can be encrypted for both data on disk and copies of records that are cached in memory. Data can be encrypted with quantum safe algorithms without z/TPF application code changes or application outages.

For more information and examples, see “The z/TPF database architecture” on page 68.

Large memory exploitation

An IBM z16™ server can have up to 40 TB of memory, and an individual LPAR (z/TPF instance) can have up to 32 TB of memory. To put that in perspective, 40 TB is 1250 times more memory than a higher end x86 server with 32 GB of memory. In addition to running more memory-intensive workloads like Java, very large memory can be used to cache an enormous amount of data and results in memory. This approach can reduce your CPU costs and transaction latency, and allow you to build smarter applications.

There are different use cases for leveraging very large memory on z/TPF and many advantages in doing so, such as the following ones:

- New workloads: Java and other memory-intensive processing can be deployed at scale.
- Caching database records: The ability to cache a massive number of database records or even entire databases in memory allows you to achieve the following goals:
  - Reduce transaction latency and CPU consumption by lowering or eliminating physical I/O operations.
  - Build smarter applications by making data instantly available for applications to use in decision making.
- Caching results: Cache the result of a calculation rather than having to continuously compute it. By doing the calculation once and then caching the result in memory for use by subsequent requests, you can achieve the following goals:
  - Reduce transaction latency, CPU consumption, and I/O by eliminating compute time and the I/O operations that are needed to obtain the result.
  - Achieve 100% cache accuracy by caching the results on z/TPF. When the results cache is on z/TPF, it is quick and inexpensive for a z/TPF application to invalidate or delete a results cache entry that becomes outdated; external cache solutions can have stale or outdated results.

For more information and examples, see "Large memory exploitation" on page 76.

z/TPF memory access

In "Large memory exploitation " on page 5, we explained the key advantages of leveraging large memory on z/TPF, but there also are performance advantages to accessing memory on z/TPF.

One differentiator is that all memory that z/TPF applications access is permanently backed by real memory. Most other operating systems implement a memory paging architecture. The disadvantage of memory paging is that if an application attempts to access virtual memory that is not in physical memory, the application process is suspended. The application is suspended while the operating system pages, or writes out, an area of virtual memory that is in physical memory to disk to free up space, which results in a write I/O. This situation frees up physical memory, and then the requested virtual memory is read from disk, which results in a read I/O. The loss of control and disk I/O when accessing memory can have a significant impact on CPU consumption and transaction response time. The z/TPF system does not use memory paging, so every memory access by a z/TPF application is to real memory, which makes it fast and efficient, and with zero I/O.

Another important performance aspect of memory usage is pre-allocated memory. On other platforms, transactions make hundreds or even thousands of "get memory" and "release memory" system calls. Across the system, these calls might result in millions of memory allocations per second, which increases the CPU overhead. With z/TPF, you can pre-allocate a configurable amount of memory for each process. When properly tuned, almost all z/TPF "get memory" system calls use memory that already is attached to the process and available, which results in lower CPU costs.

For more information and examples, see "z/TPF memory and high availability" on page 82.

Kernel-scoped sockets and middleware

Most platforms implement process-scoped TCP/IP connections, which means that a socket and any middleware that uses that socket is tied to one application process. If that process ends or fails for any reason, that network connection (socket) is terminated. Starting a network connection can be expensive in terms of CPU and latency, especially for a secure (TLS) connection, which is normally required.

On z/TPF, sockets and middleware connections are owned by the kernel and are not tied to any application process. This approach has several advantages, including the following ones:

- Reduced transaction latency and CPU consumption: By using existing connections for new transactions, you do not incur the latency and CPU overhead of starting a new connection.
- Higher availability: When loading new versions of your applications, it is not necessary to restart network connections.
- Proven ability to scale: For years, z/TPF systems have supported hundreds of thousands of active network connections, which are possible because the connections do not require their own active process. For example, if z/TPF is processing 200 transactions concurrently, then there are 200 active application processes. This situation is true whether there are 500, 5000, or 500,000 active network connections.
- Increased resiliency: A single, slow-responding remote server does not impact workloads on z/TPF that are not associated with that problematic remote server.

For more information and examples, see "Kernel-scoped sockets and middleware " on page 83.

Why Java is secure on z/TPF

Usually, Java delivers on the value claim of "write code once, run it anywhere." Because of security exposures that are encountered from running Java on other platforms, organizations might be hesitant to embrace Java on z/TPF.

Here are three key differences that enable z/TPF to support Java with the security that you need:

- Secure code load: Loading of all executable code on z/TPF is tightly controlled and can be done only by the operator, which includes Java, which on other platforms is often deployed and run by using FTP with limited safeguards in place. In addition, Java code (like `.jar` files) is stored in an area of the z/TPF file system that only certain parts of the z/TPF kernel are allowed to update, such as the z/TPF program loader.
- Immune to many high impact vulnerabilities: Error conditions, like a buffer overflow, which can result in a program gaining root access on other platforms, do not exist on z/TPF. For example, a buffer overflow (an attempt to access a memory location that does not exist) results in a dump that is followed by z/TPF terminating that application process.
- No direct access to z/TPF data: Java programs do not have direct access to your existing z/TPF databases, either older style FIND/FILE databases or the newer IBM z/TPF Database Facility Enterprise Edition (z/TPFDF) databases. On many other platforms, executable code and databases are on the file system together. In this architecture, if a defect allows a program to obtain root access, that program might access (and update) everything on that platform.

  z/TPF databases are not in the file system. Therefore, even if a Java program managed to obtain root access in the file system, that program would *not* have access to your z/TPF databases. Java code can access only z/TPF databases through code (in C/C++ and assembler language) that you define as a callable service by Java code on z/TPF. For example, if the only such service you make available to Java is to read a hotel reservation record with the person's name and check in date as input, that is the only data in any of your z/TPF databases that Java code on z/TPF can obtain. For multi-language z/TPF applications, if you have code in C/C++ or assembler language that is calling a Java program, you can pass data that way too, but again your application code in C/C++ or assembler language controls what data is visible to Java.

Continuous application modernization journey

Because the IT landscape is continually changing, as a best practice, businesses should consider application modernization an ongoing journey, not a one-time implementation. It is an initiative that maximizes the potential of existing investments. With this situation in mind, businesses can evolve and grow to continuously align with current and future business needs. The ongoing journey includes the following tasks:

- Building the right foundation

  Optimize hardware and software costs while streamlining application management and performance. Businesses can integrate the IBM Z platform into a hybrid cloud solution to simplify access to z/TPF applications. Implement a standardized approach to IT automation to help businesses support a hybrid cloud architecture in a consistent and productive way.

  Also, organizations can unlock more revenue by increasing access to data for analytics and AI through an application programming interface (API) and a data modernization strategy while still ensuring security and compliance.

- Increasing business agility and productivity

  Move to an enterprise DevOps process and an automated CI/CD pipeline. You can fully align this approach to meet present priorities while being open and standard across the business.

- Accelerating your journey

  Reduce time to value with proven application modernization patterns, tools, and best practices. These resources include how-to guides and showcases for working with z/TPF, IBM z/OS®, Linux on IBM Z, IBM LinuxONE, and public cloud together. Taking a continuous approach to modernization with the IBM Z platform alongside public cloud can be the catalyst for lowering costs and increasing return on investment (ROI).

Where do you start

In this IBM Redbooks publication, we describe several entry points where you can start or continue your z/TPF application modernization journey depending on your business needs, environment, challenges, and current processes. Select which entry points drive the most immediate value for your business. As a best practice, you should do the following tasks:

- I. Optimizing the cost and performance of existing z/TPF applications
- II. Enhancing and modernizing applications
- III. Integrating across hybrid cloud
- IV. Simplifying information sharing and data access
- V. Getting more agile with enterprise DevOps
- VI. Making AI-driven decisions at scale

I. Optimizing the cost and performance of existing z/TPF applications

IBM continued investments in its mainframe hardware and software is designed to optimize the cost and performance of running and deploying applications on z/TPF and the IBM Z platform as a whole.

This entry point describes approaches to maximize CPU resources and implement cost-effective pricing for modernization. It also guides you through the different optimizations and capabilities that you can leverage to bring down operating costs, improve the efficiency of existing z/TPF applications, and increase resiliency.

Maximizing CPU resources

One common z/TPF environment configuration is to have both z/TPF production systems and z/TPF test systems (or other workloads that are not real-time) sharing CPU resources alongside each other in the same IBM Z server. With this configuration, you can maximize your CPU resources, optimize costs, and leverage increased capacity for your test systems without impacting production workloads.

For example, Figure 3 shows an IBM Z server with 14 physical CPU engines that are shared across the z/TPF production and test systems (LPARs). The z/TPF production system has 12 logical CPU engines, and the z/TPF test system has six logical CPU engines. Because there are more logical CPU engines than physical engines, priority is assigned to the LPARs to determine which LPARs get to use more of the physical CPU resources when the IBM Z server is nearly 100% CPU busy. Assigning a higher priority to the production system is recommended so that it can use as many CPU resources as it needs. In this example, the production system can use up to 12 physical CPU engines whenever it needs to, and the test system can use the remaining CPU resources. The z/TPF test environment always gets at least two physical CPU engines, but can increase up to six physical CPU engines depending on the production workload.

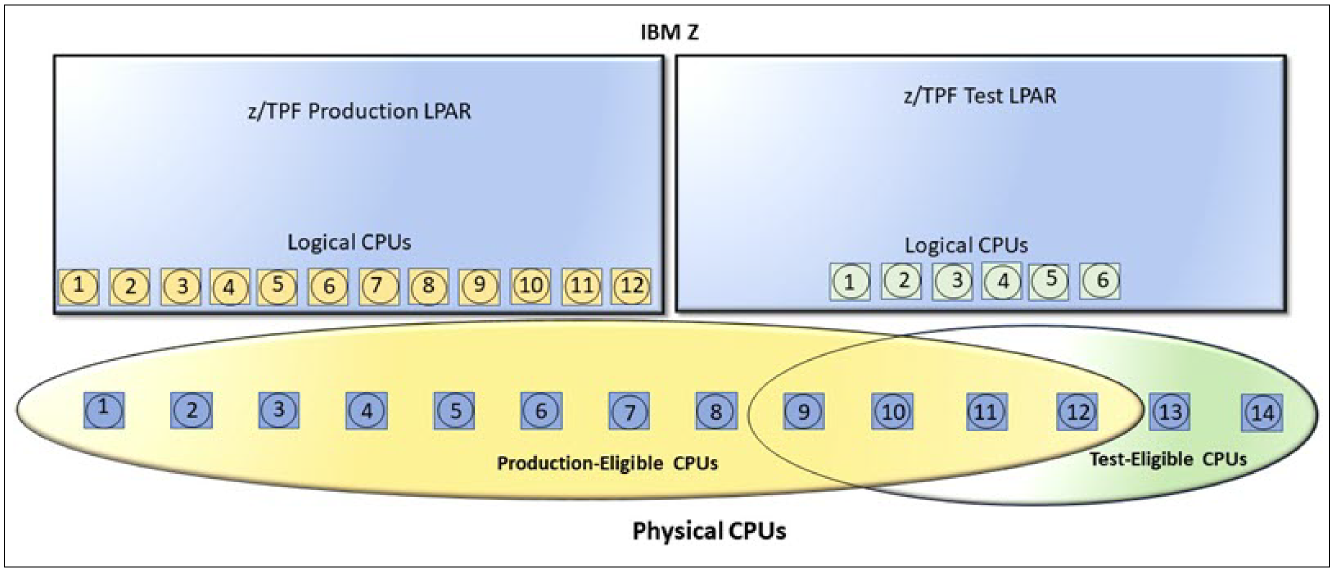


Figure 3 Sharing CPU resources across workloads

Leveraging on-demand CPU capacity

Many z/TPF environments have workload spikes a few days a year, such as an airline reservation system when there is bad weather in multiple cities, which requires significant rebooking processing. Another example is credit card processing systems when retailers are offering major sales. By leveraging z/TPF Dynamic CPU support, you can seamlessly handle unexpected workload spikes by issuing a single z/TPF operator command to add CPU capacity to your production system for the duration that it is needed. This capability allows you to pay only for the days that the extra capacity is enabled, which provides a cost-effective, high-performance solution to manage infrequent yet high-value workload spikes.

For example, a production system normally uses up to eight CPU engines, but a handful of days per year it needs as many as 12 CPU engines to accommodate significant increases in workload. With z/TPF Dynamic CPU support, you can start your z/TPF system with 12 logical CPU engines (I-streams) defined, even though you are entitled to use only eight engines for transactions.

Figure 4 shows an example of this environment. The I-stream cap is set to 8, meaning that eight engines are used to process transactions. The other four engines are fenced, which means that they are not enabled to process workload, but they are initialized and ready to be used when you encounter unexpected workload spikes and choose to enable one or more of those extra four engines.

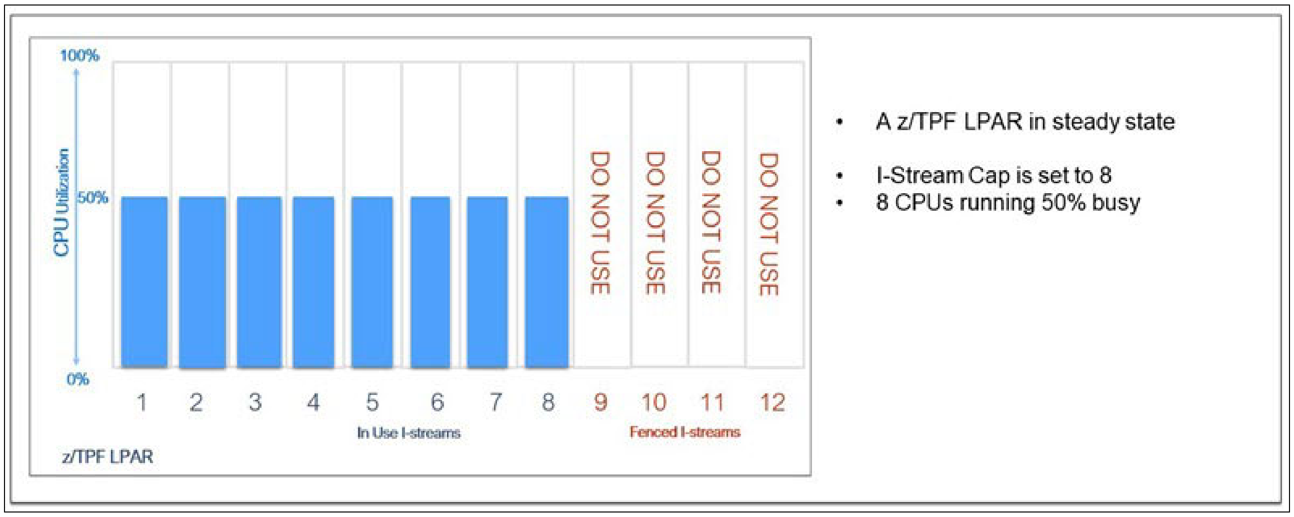


Figure 4 Dynamic CPU

Figure 5 shows an increase in workload where the eight engines processing transactions become 95% busy, and projections indicate that the workload volume will increase even higher. Therefore, to avoid the impacts that occur if a system becomes 100% busy, the z/TPF operator issues a command to activate more I-streams to unleash extra needed CPU capacity.

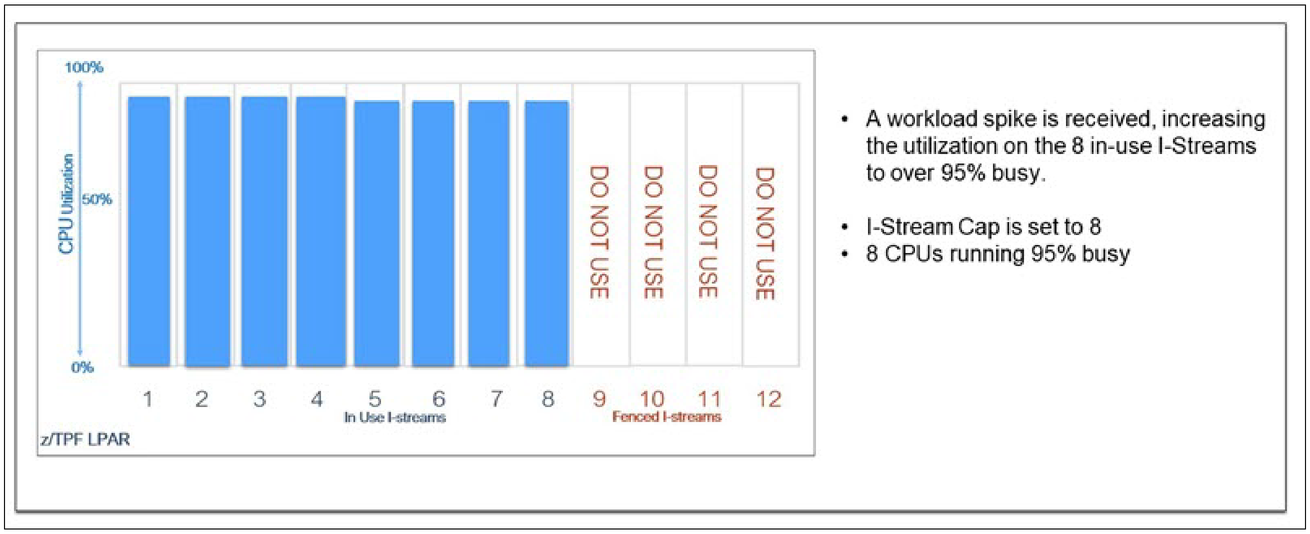


Figure 5 Dynamic CPU and increased workload

Figure 6 on page 11 shows that the operator increased the I-stream cap from 8 to 11, allowing three more engines to process transactions. The 11 engines processing transactions are now back to a lower level. In this example, they are back to only 70% CPU busy, which allows the system to safely handle the increased workload.

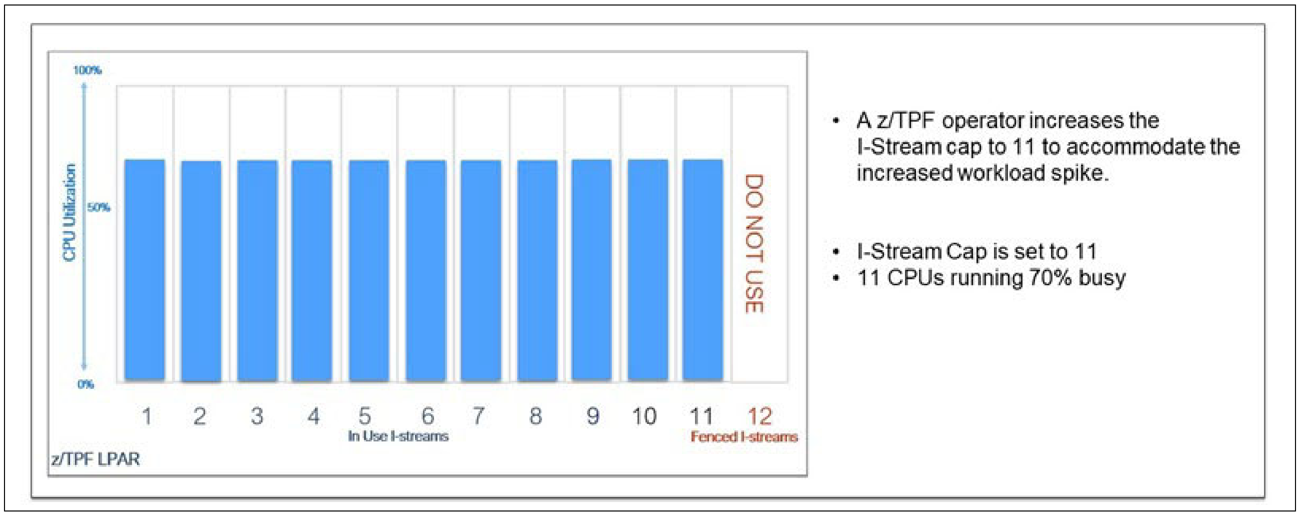


Figure 6 Dynamic CPU: Adding more capacity

Figure 7 shows that on the next day when the extra CPU capacity is no longer needed, the z/TPF operator issues another command to lower the I-stream cap back to the standard, entitled eight engines.

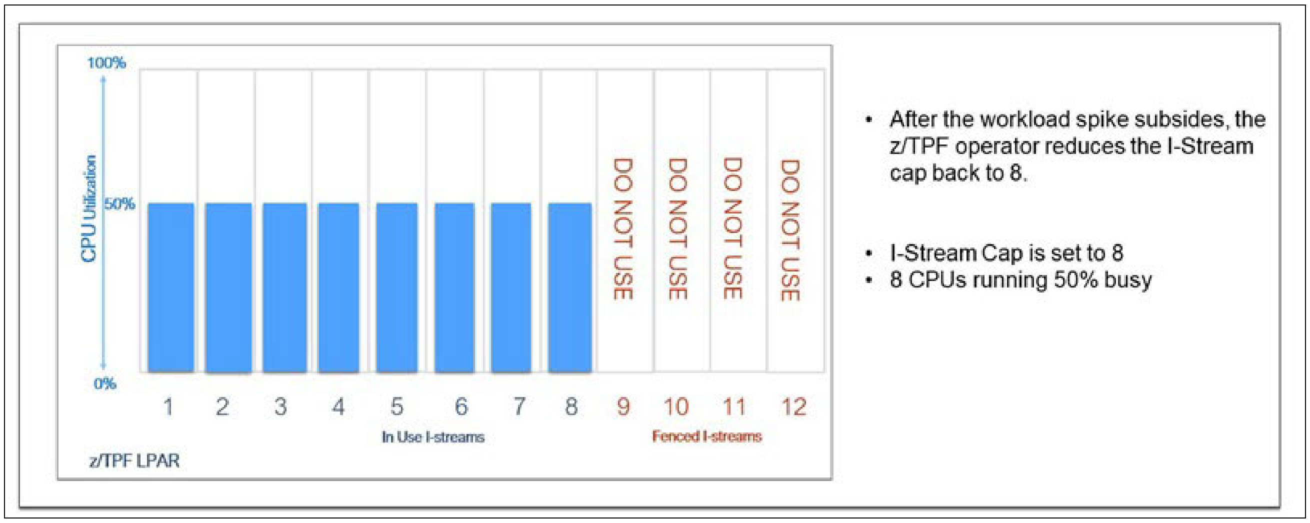


Figure 7 Dynamic CPU

For z/TPF production systems sharing CPU resources (shared PR/SM), the standard IBM Z CPU On/Off Capacity on Demand (OOCoD) process can be used to maximize CPU resources and optimize cost, similar to z/TPF Dynamic CPU support. In this environment, the z/TPF operator follows the OOCoD process to add more physical engines to the pool that is shared by multiple LPARs. They need to issue only a single z/TPF command to increase or reduce the I-stream cap and CPU capacity of the z/TPF system.

Leveraging platform performance improvements

IBM delivers continuous z/TPF performance improvements based on client usage patterns and workload growth. Often, enhancements are transparent to applications and can provide immediate value when they are installed. These enhancements include improvements to middleware (REST, HTTP, and IBM MQ), TCP/IP stack, task dispatcher, database utilities, Java, and resource or lock contention. Some of the latest z/TPF improvements that can be leverage include the following ones:

- More efficient system operations by leveraging low-priority utilities.
- Reduced CPU, I/O, and network costs with hardware compression (IBM z15® and later).

In addition to software currency, performance benefits can be realized through hardware currency and leveraging the latest IBM Z platform innovations. For example, key z/TPF improvements on IBM z16 include the following ones:

- Increased hardware capacity to encrypt all data at rest, in flight, and in use by using quantum safe symmetric cryptography.
- 50% more memory cache capacity that is available to each CPU core.
- On-chip Artificial Intelligence Unit (AIU) accelerator for AI frameworks running on colocated Linux on IBM Z.

Reduced cost of modernized applications with specialty-priced processors

z/TPF offers a specialty-priced processor that helps businesses lower the cost of running new functions and applications, and extending and enhancing existing applications on the platform while allowing them to leverage z/TPF key differentiating strengths in performance, security, availability, and scale.

IBM z/TPF Transformation Engine (TE) is a specialty-priced, general-purpose engine that does not impact the software usage charges of existing applications. TEs provide a cost-effective way to extend and modernize existing applications and add new applications to the z/TPF platform.

A TE is designed for newer, heavier weight processing that typically is used when modernizing applications, such as Java, REST, HTTP, XML, JSON, and TLS. The z/TPF system calculates what percentage of active time was in regular code versus TE eligible code and reports those results for accounting and billing purposes. The potential total cost of ownership (TCO) savings can be considerable compared to running the same application on regular general-purpose processors, x86 servers, and public cloud infrastructures.

Also, with z/TPF as an integrated part of your broader IBM Z platform, extra specialty-priced processors can be leveraged to lower TCO while maintaining key platform strengths:

- IBM Z Integrated Information Processor (zIIP) for z/OS applications
- IBM Integrated Facility for Linux (IFL) for Linux application workloads on IBM Z and IBM LinuxONE

Optimizing costs and increasing resiliency on z/TPF as applications change

Many z/TPF test environments are 10 - 20% of the size of production due to cost constraints. Certain types of code defects or application design flaws show up only at high transaction volumes, so there is a need to test code at scale before deploying significant application changes to production.

It is not cost-effective to purchase extra capacity permanently to support a larger test environment. IBM Z Business Resiliency Stress Test (zBuRST) allows you to temporarily increase the CPU capacity of your z/TPF native test environment by using spare IBM Z physical resources to load and stress test at production scale. zBuRST provides flexibility to optimize cost and resiliency by allowing you to purchase the number of tokens that is necessary based on how many days per year that you must do full-scale tests.

Figure 8 on page 13 shows a z/TPF production system that has 20 CPU engines and a z/TPF native test system that has four CPU engines. This test environment is normally used for performance and stress testing, and it is only 20% the size of the production environment. When zBuRST is activated, the capacity of the z/TPF native test system expands to 20 CPU engines, enabling full-scale testing. When those tests are complete, zBuRST is deactivated to return the test system to its original size of four CPU engines.

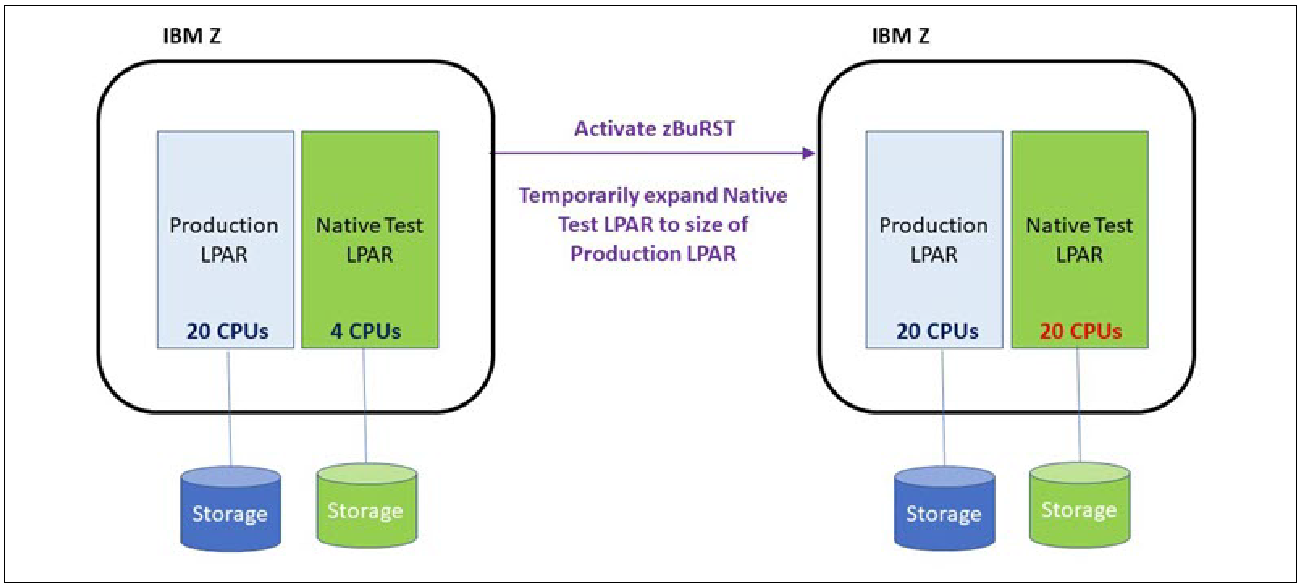


Figure 8 Temporarily expanding the capacity of a z/TPF native test system for full-scale testing

ZBuRST also provides the flexibility to use spare, existing resources to support full-scale testing. For example, there might not be enough direct access storage device (DASD) processing capacity at the primary data center to handle both the production workload and the full-scale test system, yet one or more disaster recovery (DR) data centers have CPU and DASD hardware available. With zBuRST, you can use the idle DR hardware periodically to do full-scale production volume testing before you deploy new or updated applications into production, as shown in Figure 9.

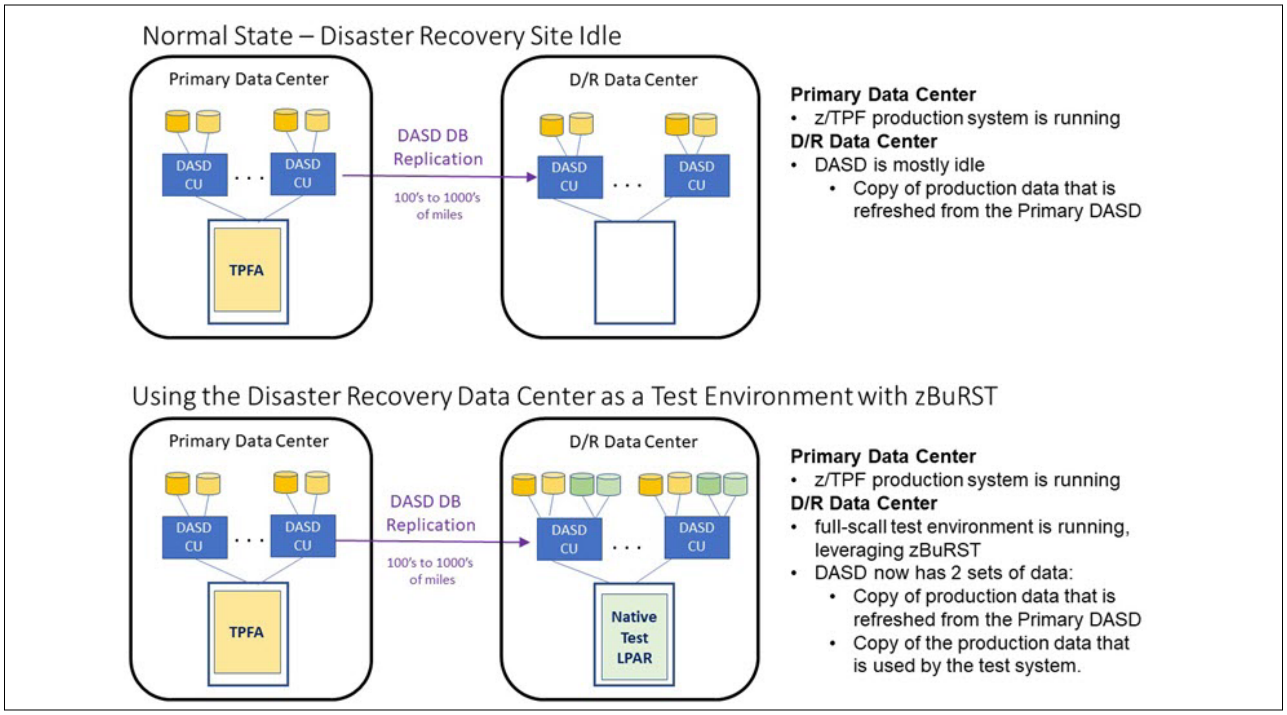

Normal State – Disaster Recovery Site Idle
Primary Data Center
DASD CU
DASD CU
TPFA
DASD DB Replication
100's to 1000's of miles
D/R Data Center
DASD CU
DASD CU
Primary Data Center
• z/TPF production system is running
D/R Data Center
• DASD is mostly idle
• Copy of production data that is refreshed from the Primary DASD
Using the Disaster Recovery Data Center as a Test Environment with zBuRST
Primary Data Center
DASD CU
DASD CU
TPFA
DASD DB Replication
100's to 1000's of miles
D/R Data Center
DASD CU
DASD CU
Native Test LPAR
Primary Data Center
• z/TPF production system is running
D/R Data Center
• full-scall test environment is running, leveraging zBuRST
• DASD now has 2 sets of data:
• Copy of production data that is refreshed from the Primary DASD
• Copy of the production data that is used by the test system.

Figure 9 Production volume testing with zBuRST

Improving resiliency with flexible capacity

Overall business continuity requires operations to run continuously, even in a DR situation. The IBM Z Flexible Capacity for Cyber Resiliency offering, which was introduced with IBM z16, can be leveraged to shift capacity between participating IBM z16 systems at different sites for up to 12 months. This solution allows organizations a flexible, cost-optimized approach to increase resiliency, and in many industries such as banking, meet regulatory requirements. For more information about moving z/TPF workloads from one data center to another one, see the reference architectures for "Single-image z/TPF systems" on page 48 and "Active-active z/TPF cluster" on page 50.

Adopting usage-based pricing

With usage-based pricing, your z/TPF software bill is based on how much CPU z/TPF consumes rather than based on the total size of the IBM Z server. This approach allows you to consolidate multiple workloads into fewer IBM Z servers to reduce your data center footprint and maximize the CPU resources in those servers.

II. Enhancing and modernizing applications

Enterprises need their applications to be maintained, extended, deployed, and managed in a way that meets market demand and allows their businesses to grow, including continuously adding new capabilities. How these enhancements are made and where the applications are deployed depends on the type of application and the nonfunctional requirements around quality of service, security, and accessibility to data and maintainability.

This section describes the next entry point to your continuous application modernization journey, and the best practices for building new capabilities and where they are best deployed.

New ways of development

Over time, many z/TPF applications have become large, with thousands to even tens of thousands of program segments. Enhancing these large monolithic applications by using "big bang rewrite" approaches or outdated "waterfall" development methods can introduce significant risk and requires expert-level skill. In addition, the testing that is required for changes to these applications is slow, complex, and resource-intensive. For all these reasons, new ways of development are needed to modernize your applications with reduced risk and more rapid return on investment for your business.

The guiding principles to approach development for application modernization are as follows:

- Break monolithic applications into smaller, reusable services.

  By using tools such as z/TPF Message Analysis Tool, you can more quickly discover opportunities to break down large applications into smaller business functions.

- Invest in automated testing.

  By creating repeatable automated test cases, you can ensure the quality of modernized applications and create a reusable regression test suite for future enhancements that you make to these applications.

- Progressively modernize applications or services to target high-value areas and achieve a quick return on investment.

  By focusing on discrete parts of the application, instead of a full rewrite, you can reduce risk and achieve faster time to value.

- Leverage the choice of technology for the new or enhanced services.

  z/TPF offers mixed-language support, so you can modernize by using the programming language that is best suited for the business function and development team. You can also leverage open source (especially Java) for faster time-to-market with fewer resources.

Modernizing applications on z/TPF

Progressive application modernization should begin with applications and services that provide the fastest time to value. Depending on your business and application landscape, high-value areas might be identified as code that is modified most frequently, code that has historically had a high number of defects, or code that is poorly written and takes a long time to understand and modify. For example, you have an application that is made up of 200 program segments. In the last 3 years, 90% of the defects for that application were in 12 program segments, so you quickly and significantly can improve quality by refactoring that 6% of the application code base.

After you determine which applications to modernize, follow these best practices:

- Discover and analyze existing applications.
- Select a target platform by using fit-for-purpose.
- Modernize by using the right programming language.

Discovering and analyzing existing applications

Application understanding and discovery play a key role in modernization on z/TPF, especially for applications that might have been designed decades ago where tacit knowledge has been lost. By approaching this effort with the right tools, you can reduce the scope and complexity, and accelerate the process of discovering existing applications that can be decomposed into smaller services where necessary.

Reducing scope and cost by identifying dead code

Especially for older applications that have grown in size over time, it is important to first focus on removing dead code that might be adding complexity and scope. For example, certain message types and formats, options on messages, and communication channels might no longer exist. By first analyzing your application to eliminate dead code, you can simplify your modernization effort.

Because of the monolithic nature of many z/TPF applications and intertwined call patterns where applications within z/TPF call each other by using various entry points, static analysis is of limited value. Instead, the z/TPF Active Program Detection (APD) tool allows you to identify possible dead code by analyzing which programs are used (called) on your z/TPF production system. There is virtually no overhead from running APD. By running APD over a period of time on your production system, it identifies a list of application programs that have never been called, which are potentially obsolete. A subject matter expert (SME) can use this list to confirm which programs can be eliminated.

Within active programs, there might still be an opportunity to remove dead code. By using the z/TPF scriptable code coverage tool, you can see which source lines in specific application program segments ran, which means that the lines of source code that did not run might be dead code. Identifying and eliminating dead code from your application code base reduces the scope and cost of your application modernization effort, and the cost of enhancing those applications.

Understanding the application

After you identify the application code base that is still in use and must be maintained, one key need is training new developers on what the applications do. Static analysis is of limited value here because often the input to the application is an input message that can contain zero or more instances of element 1, zero or more instances of element 2, and so on. In addition, there are hundreds of possible values for certain fields in an element instance, like an airport code. In other words, theoretically there are millions of possible input message combinations that can drive the application code paths in different ways. In reality, only a fraction of possible input message combinations is ever seen.

The z/TPF Message Analysis Tool allows you to collect data on real production system transactions and show you the code path that each transaction collected took going through your applications. This tool enables new developers to better understand the application, such as the application logic and what databases were accessed and updated. This information can also be used to decide whether and how the application may be broken down into smaller functions and services.

Selecting a target platform by using fit-for-purpose

Often, there is a choice about where to deploy applications. At a high level, the best fit for a cloud environment often is customer-facing experiences and digitalization efforts. Conversely, secure, high-volume transaction processing and access to real-time data are a best fit on the IBM Z runtime environment, including cloud-native applications, traditional z/TPF workloads, and distributed applications. Making the best decision for your business requires understanding the fundamental value of z/TPF (see “Advantages of the z/TPF architecture” on page 3), the requirements of the application itself, and the acceptable degree of latency.

Sometimes, applications might have requirements, such as security or availability, that narrow down the options for the best-fit platform. However, the decision is often based on the acceptable degree of latency, which can be more complex. In addition to physical distance between client and server nodes, variables like transaction volume and the need to access or update data can compound the impact of latency. Failure to account for the impact of latency in your fit-for-purpose analysis can have a significant business impact, such as not being able to meet response time requirements of SLAs, preventing growth because the transaction volume cannot scale in the environment or increasing TCO.

If two applications communicate with each other and there are multiple calls back and forth that process one transaction, the location of those applications relative to each other and the latency of each call is a major factor regarding the transaction response time. Figure 10 shows the case where both applications are running on the same z/TPF server and makes dozens of calls back and forth while processing a transaction. Because all these calls are optimized local calls within the same application process, there is no loss of control when Application 1 calls Application 2 or when Application 2 returns to Application 1. The result is an ultra-low transaction response time, which is 1 ms in this example. There is only one request and reply pair of flows through all the protocol stacks (REST, HTTP, TLS, and TCP/IP), so there is minimal CPU overhead.

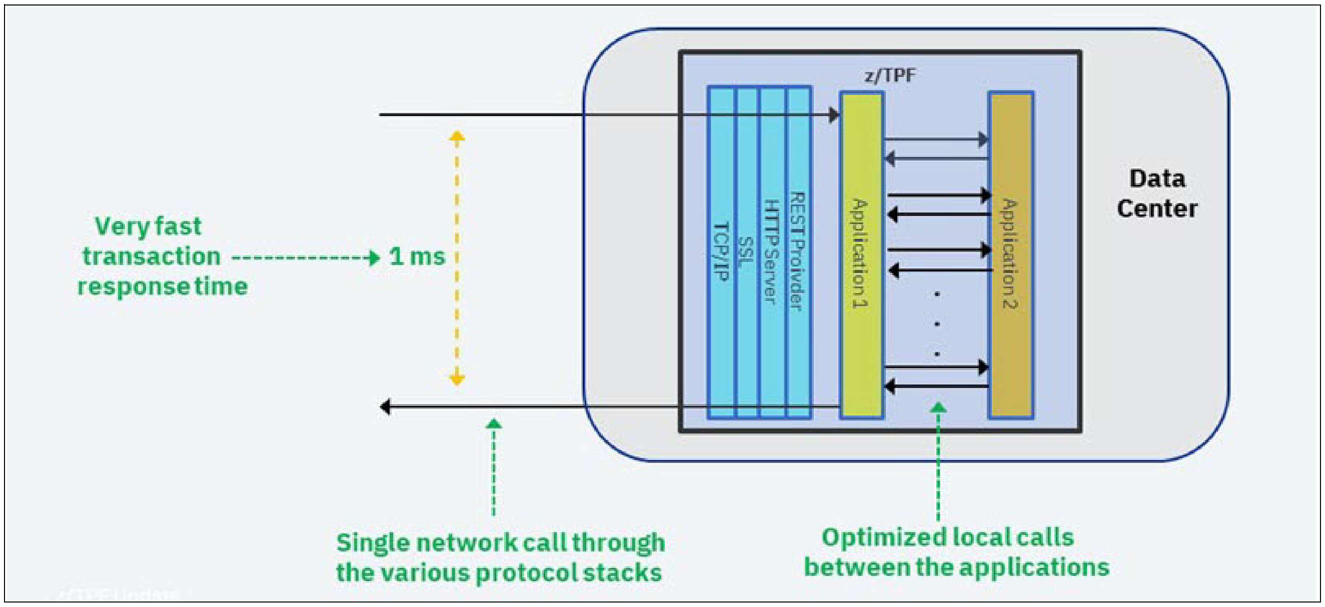

z/TPF
TCP/IP
SSL
HTTP Server
REST Proivder
Application 1
Application 2
Data Center
Very fast transaction response time
1 ms
Single network call through the various protocol stacks
Optimized local calls between the applications

Figure 10 Lowest latency and excellent performance when both applications are on z/TPF

Figure 11 shows the case where one application is moved from z/TPF to an on-premises private cloud. The dozens of calls back and forth that are processing a transaction result in dozens of external network flows and dispatching and re-dispatching the applications on both sides. Therefore, there is a higher latency and transaction response time (20 ms in this example). The dozens of request and reply pairs flowing through all the protocol stacks (REST, HTTP, TLS, and TCP/IP) on both z/TPF and the cloud server result in higher CPU costs.

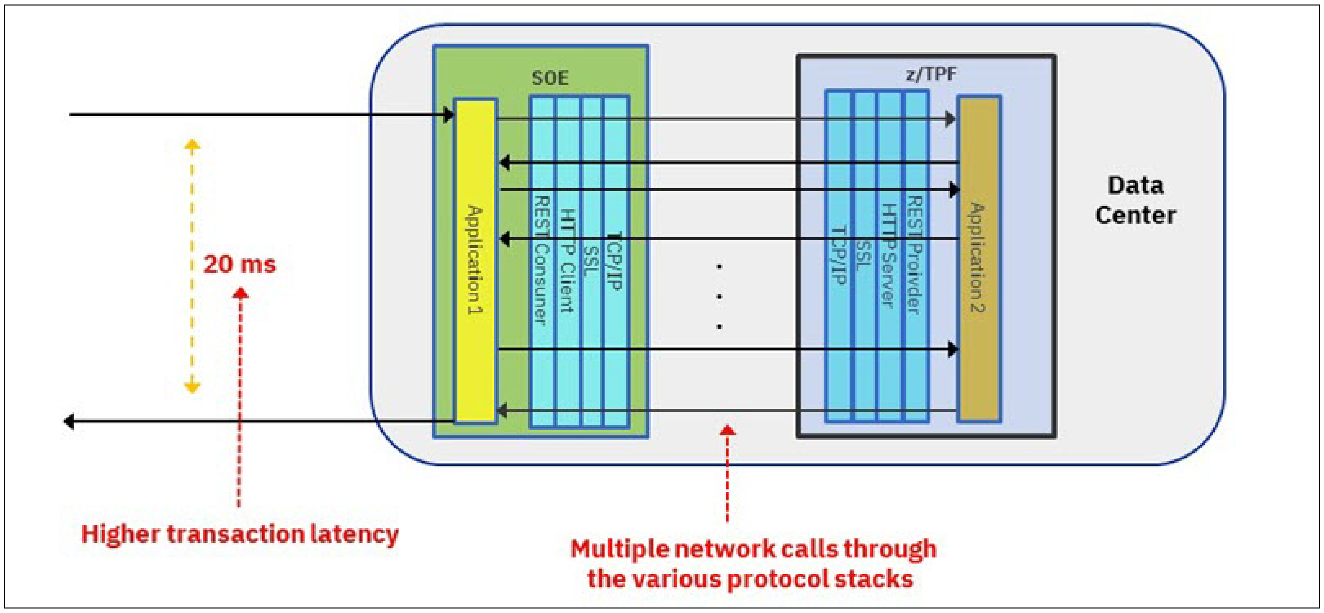


Figure 11 Increased latency and CPU consumption when an application is split on-premises

Figure 12 shows the case where one application is moved from z/TPF to a public cloud. The dozens of calls back and forth processing a transaction result in dozens of external network flows over long distance, dispatching and re-dispatching the applications on both sides. Therefore, there is a higher latency and a longer transaction response time (400 ms in this example), which might not be acceptable based on your SLAs. The dozens of request and reply pairs flowing through all the protocol stacks (REST, HTTP, TLS, and TCP/IP) on both z/TPF and the cloud server result in higher CPU costs compared to when both applications are on z/TPF.

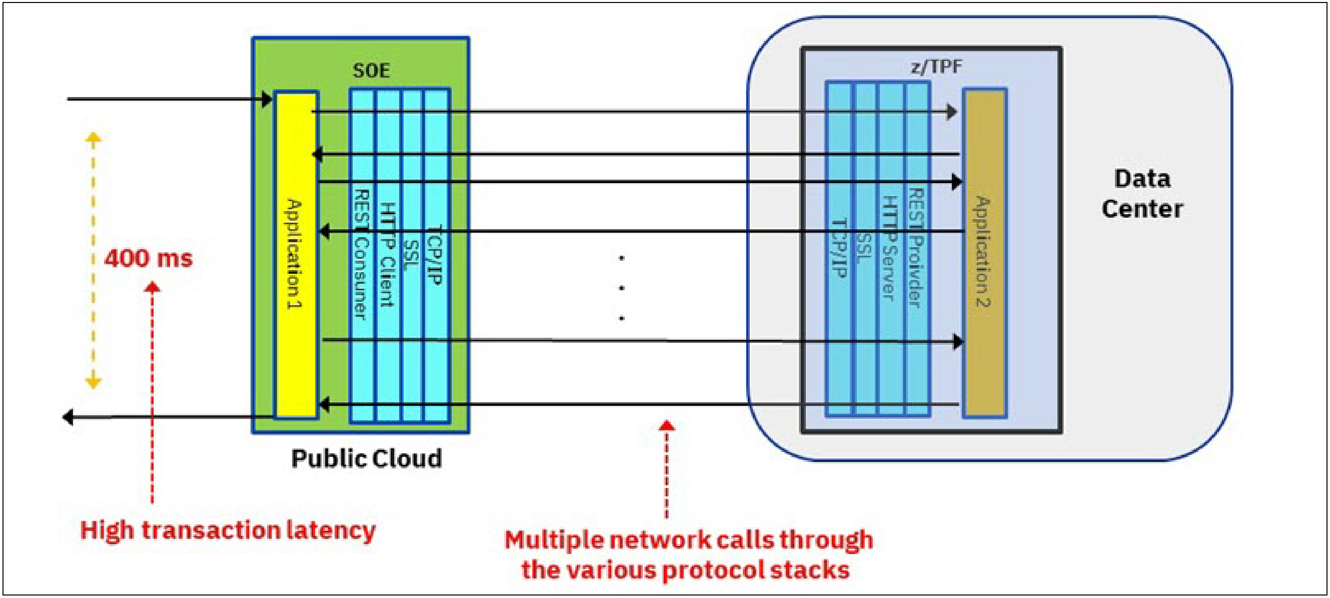


Figure 12 High latency and increased CPU consumption when applications include a public cloud

Figure 13 shows a generalized view of how the degree of latency varies regarding an application's environment. Consider the following fit-for-purpose scenarios for a new or modernized application:

- ► If the application does not require low latency when accessing z/TPF data, the best-fit platform might be public or private cloud.
- ► If the application requires low latency and has “data gravity” toward z/TPF, the best-fit platform often is Linux on IBM Z colocated on the same server as z/TPF, or Red Hat OpenShift on Linux on IBM Z located next to z/TPF on the IBM Z platform. In this environment, businesses can leverage popular open-source packages, Linux applications, IBM software, and third-party software together with z/TPF applications and data.
- ► If the application is written in Java or C/C++ and either deals with z/TPF data gravity or makes multiple calls to services on z/TPF, the best-fit platform might be z/TPF for the best performance and TCO.

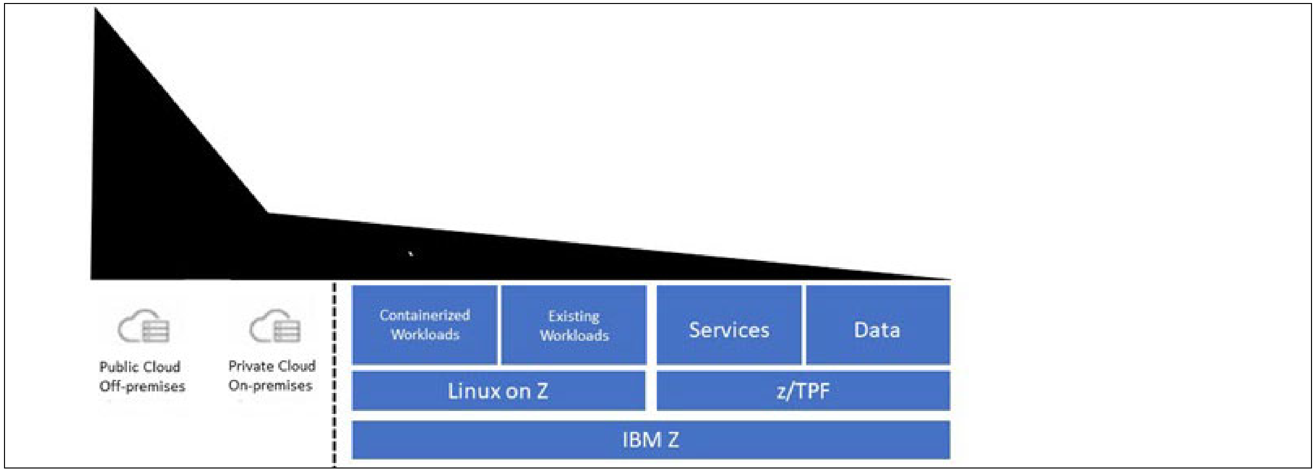


Figure 13 Latency reduces as an application moves to the right

To successfully choose the best-fit environment for your applications, it is important to understand the relationship between key attributes of your application and latency while considering both your business today and its future growth.

Transaction volume

There are fundamental architectural advantages of z/TPF that enable high-volume applications and data that is colocated on z/TPF to scale up to hundreds of thousands of transactions per second, while still maintaining SLAs (see “Advantages of the z/TPF architecture” on page 3”). In a distributed environment, thousands of application server nodes and hundreds of database server nodes are needed to handle larger workload volumes. Processing a transaction in a distributed environment typically requires many flows between nodes, which each involve multiple protocol stacks (REST, HTTP, TLS, and TCP/IP) on both the client and server nodes. With high-volume workloads, this processing results in higher CPU costs in addition to the increased latency. The latency is partly due to the extra process dispatching, but is mainly a result of networking flows. Therefore, the latency is even higher if client/server nodes are over a longer distance, such as on-premises to public cloud.

Access to core functions and real-time data on z/TPF

Often, z/TPF applications require access to core functions and real-time data (data gravity), typically with low response times that are specified by SLAs. When these applications are deployed off-platform, such as distributed or public cloud, they might encounter an order of magnitude higher latency or worse while accessing core functions and data on the IBM Z platform. This increased latency can result in lower throughput due to the frequent access to core functions and data.

Updating database records

For some workloads, a bigger problem than network latency is the increased transaction delays due to lock contention. These delays can lead to a failure to meet SLAs and prevent workloads from scaling up to higher transaction volumes. For example, a certain percentage of transactions for a workload must lock a database record while the transaction is being processed to serialize updates to that database record.

Depending on the environment, the lock duration and the ability to scale a transaction volume can be orders of magnitude better when z/TPF is chosen for the application instead of a cloud environment. Here are examples of environments and their impact on the ability to scale transaction volumes:

- On z/TPF only: Figure 14 shows the applications that access z/TPF data are running on z/TPF. The database record is locked for a short amount of time (2 ms). You can scale up to 500 transactions per second that must access and update this database record.

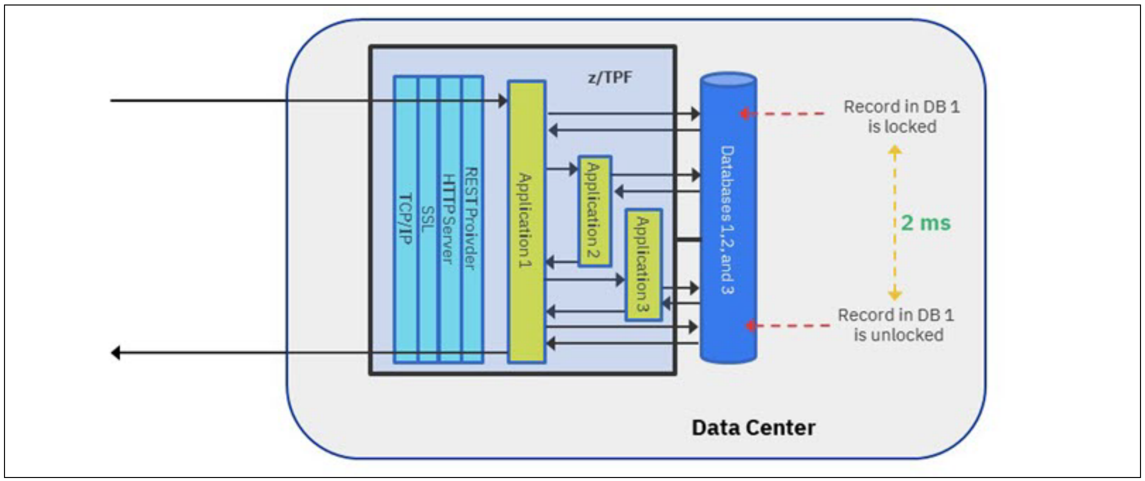


Figure 14 Performance of scale when an application is on z/TPF

- On z/TPF and colocated (on-premises) cloud: As shown in Figure 15, some of the applications that access z/TPF data are running on an on-premises private cloud. The latency increases, which causes the database lock to be held longer (in this case for 20 ms). You can scale only up to 50 transactions per second that must access and update this database record.

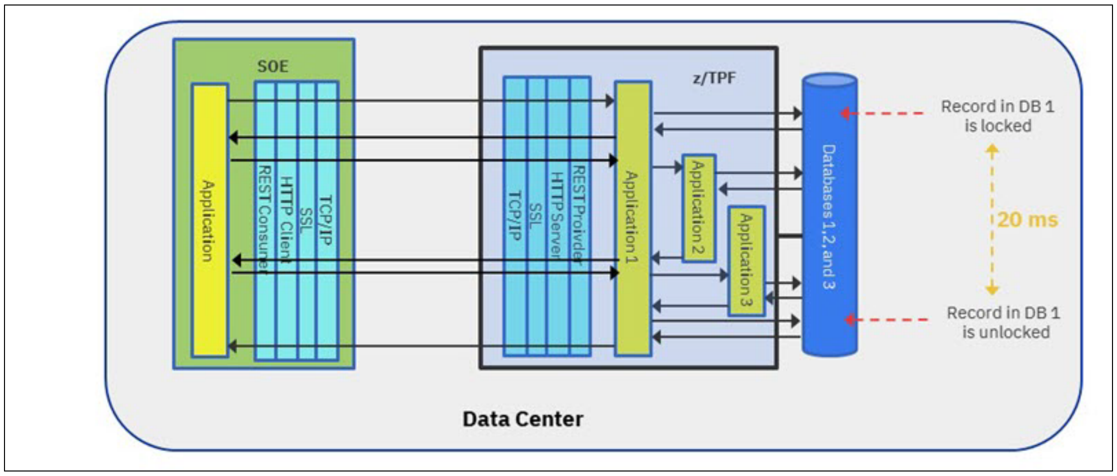


Figure 15 Increased latency and lock hold time when an application is split on-premises

- On z/TPF and partly in a public cloud: As shown in Figure 16, some applications that access z/TPF are running in the public cloud. Latency increases dramatically, which causes the database lock to be held for 200 ms. You can process only five transactions per second that must access and update this database record.

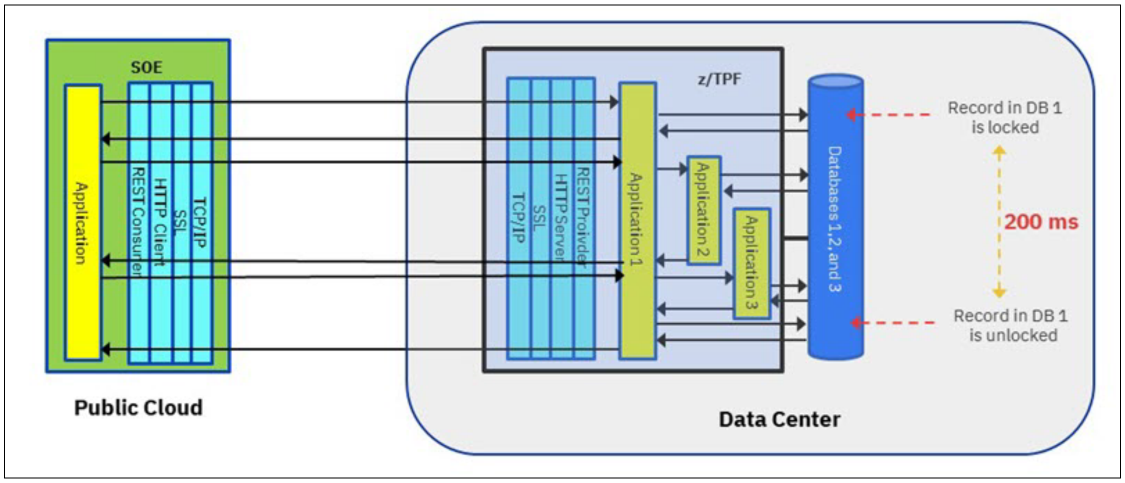


Figure 16 High latency and lock hold time when an application includes public cloud

An environment architecture with excessive database lock contention can prevent workloads from scaling up to higher transaction volumes, which is a challenging obstacle to overcome that cannot be solved with more hardware. Instead, to resolve this issue, you must perform either of the following actions:

- Make application architecture changes to reduce the rate at which a database record is locked.
- Reduce the amount of time between each database lock and unlock operation, which requires either reducing the amount of application logic that is performed under lock or reducing the latency of the application logic that runs under lock by colocating all the code.

Modernizing by using the right programming language

After the fit-for-purpose analysis, businesses choose the right programming language for their new or updated application. z/TPF provides interoperability between assembler language, C/C++, and Java. z/TPF is designed so that transactions can flow seamlessly between languages so that you can modernize by using the right programming language for each business function, where you choose either the existing application language (assembler language or C/C++) or Java.

Figure 17 on page 21 shows a z/TPF application that has parts that are written in assembler language, C/C++, and Java, where one transaction is processed by using code that is written in all those programming languages.

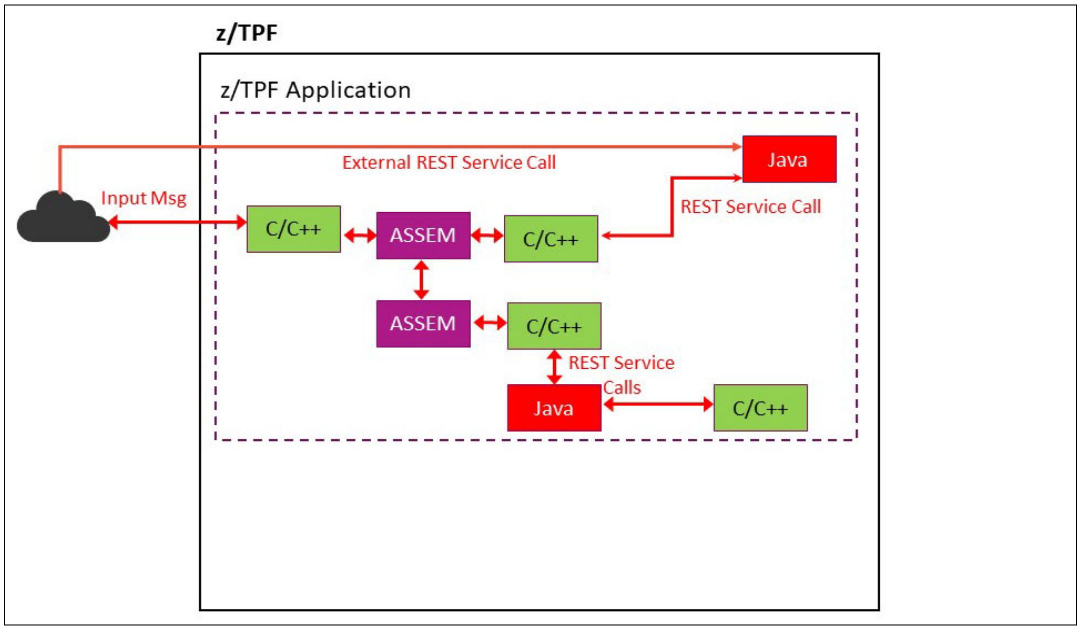


Figure 17 One transaction that is processed by multiple programming languages

Java support on z/TPF was designed to maintain a user experience that was natural for both the C/C++ developer and the Java developer:

- C/C++ developers issue C function calls to call code within that program and invoke code in other programs. If the program that is called is in Java, the necessary C structure to Java object conversion is done by z/TPF, which is transparent to the user. When the Java program gains control, it appears as though a REST API were issued that caused the Java program to be invoked. The Java program does not know that the caller is a local C/C++ program and that no REST API was issued. Processing REST APIs is natural to a Java developer.
- Java developers issue REST APIs to consume services. If the definition of a called service is that the target is a local C/C++ program, z/TPF performs the necessary Java object to C structure conversion, and then makes a local call to that C/C++ program.

There are many benefits to choosing Java for application modernization on z/TPF. For example:

- Both C/C++ calling Java and Java calling C/C++ use efficiently optimized local interfaces for best performance. As described in “Reduced cost of modernized applications with specialty-priced processors” on page 12, Java applications are z/TPF Transformation Engine (TE) eligible, which can be leveraged for cost optimization.
- Another significant benefit is the high availability of Java developer skills in the marketplace. Writing a new function in Java can enable common developer skills to be leveraged, and over time significant and relevant portions of the applications can be built in Java.
- Also, thousands of open-source Java packages are available and can run on z/TPF, including Kafka Producer, Jakarta mail, MongoDB client, and Apache Mina SFTP server. The only exception for Java open-source package use is in the rare case where it contains native code that requires a specific architecture like x86 or Windows.

Using fit-for-purpose analysis, if the decision is made that an application or part of an application should run on Linux, the new business function can be built or augmented by other language choices in addition to Java. For example, choices can include Node.js, Go, and Python. Developers can choose the language that fits their needs and leverage package ecosystems to possibly speed up delivery.

Figure 18 shows a hybrid cloud environment where a workload is processed by applications on z/TPF and applications on Linux by using some of the programming languages that are supported on those platforms.

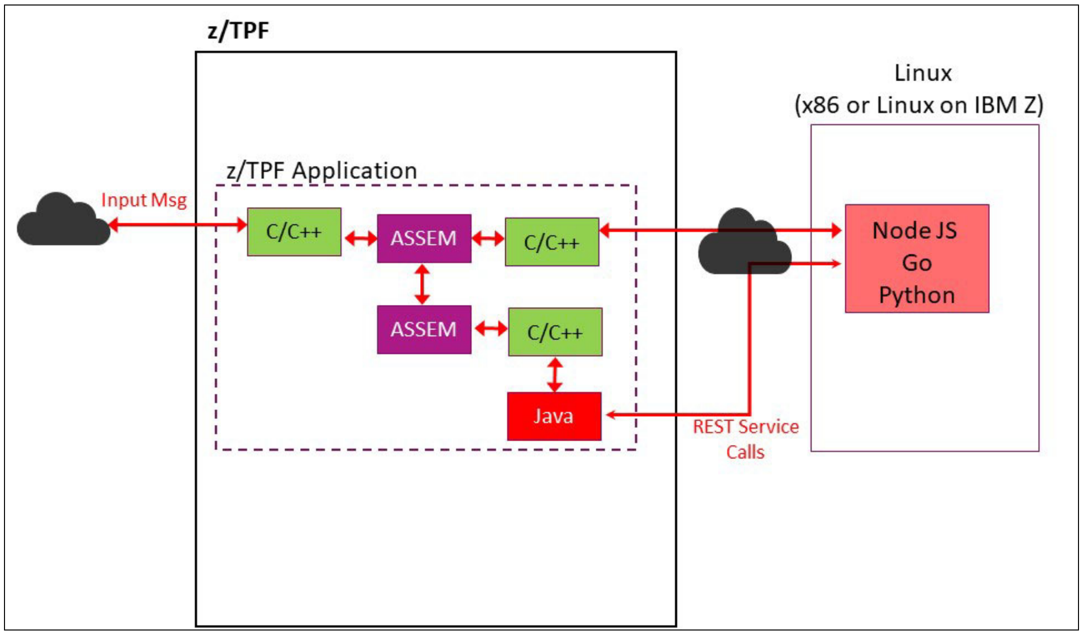


Figure 18 Hybrid cloud: Workload that is processed by applications on z/TPF and Linux

III. Integrating across hybrid cloud

With a hybrid cloud environment, businesses must be able to integrate applications across the IBM Z platform and the cloud. Today, one of the most secure and efficient ways to expose and consume application services and data is by using APIs. Business-critical z/TPF applications can and should be a part of an enterprise API strategy so that modernized applications can leverage the security of the IBM Z platform while integrating applications into the digital services on the cloud. In addition, by leveraging z/TPF as part of an event-driven architecture, businesses can provide customers with real-time or near real-time data on demand. Examples include seeing the status of a payment, getting updated arrival time information for a flight, or tracking your checked luggage.

Using APIs

APIs are a pervasive part of a digital ecosystem because they simplify access between applications. Open APIs are the standard for software to securely access and update low-volume business data on demand in real time.

To optimize your API ecosystem, consider that the platform hosting the API service impacts the reliability, scalability, throughput, and security of the API. The IBM Z platform, including z/TPF, is built to provide high value in all these areas. With z/TPF, you can accomplish the following tasks:

- Create consumable APIs to make z/TPF applications and data accessible to your enterprise and cloud applications.
- Call APIs from z/TPF applications to extend them with data from internal enterprise APIs or public APIs from cloud services, for example, a weather or currency conversion service.

Externally, the IBM Z platform is no different from any other platform in the implementation and deployment of APIs. Figure 19 shows services on z/TPF, other IBM Z operating systems, and a private cloud, all of which are accessed through a centralized API Management (APIM) gateway by public and private cloud service consumers.

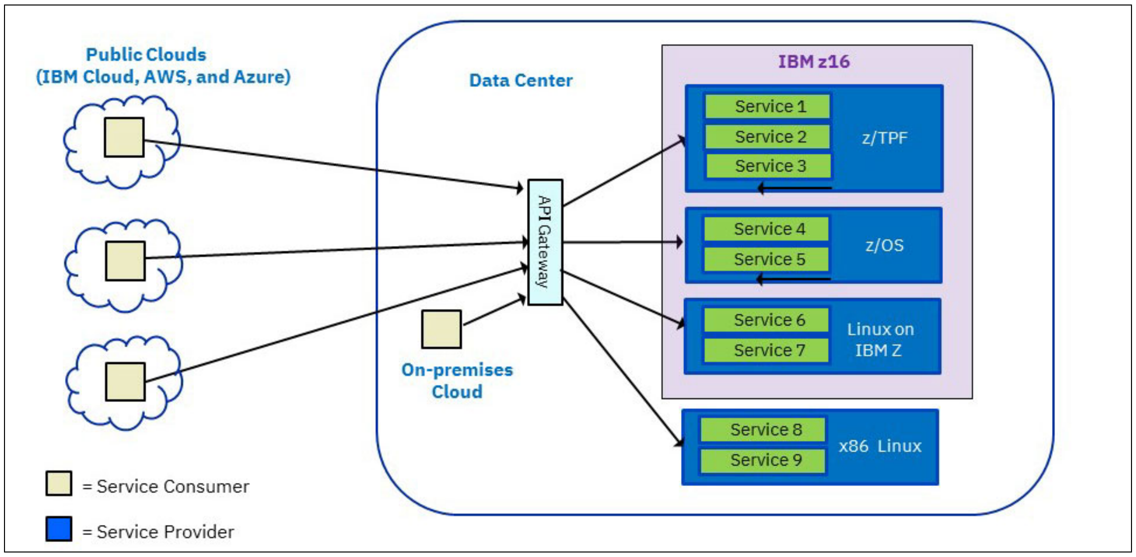


Figure 19 Centralized API Management for all services

Many z/TPF applications were created decades ago, enhanced hundreds of times since then, and have been accessed by using many communication methods like SNA, IBM MQ, HTTP, several user proprietary protocols, and most recently REST.

Before REST, a common architecture was to separate the z/TPF application from the communication method because the business logic (the application) needed only the relevant data. Therefore, no changes to the application are needed when an extra communication method is added.

For example, the interface to the application might be a C structure that contains the input data that is needed to process the transaction and the output is another C structure that contains the return code and output data. For example, the input to a flight availability service might be a C structure containing the origin and destination airport codes, date, and number of passengers. The output data contains the return code and a list of flights.

Each communication method has wrapper code that parses and pulls the necessary data from the input message, builds the C structure that is the input interface to the application, invokes the application program, and then builds the output message based on the data that is returned from the application.

Figure 20 shows remote clients that use three communications methods to access the two applications on this z/TPF system.

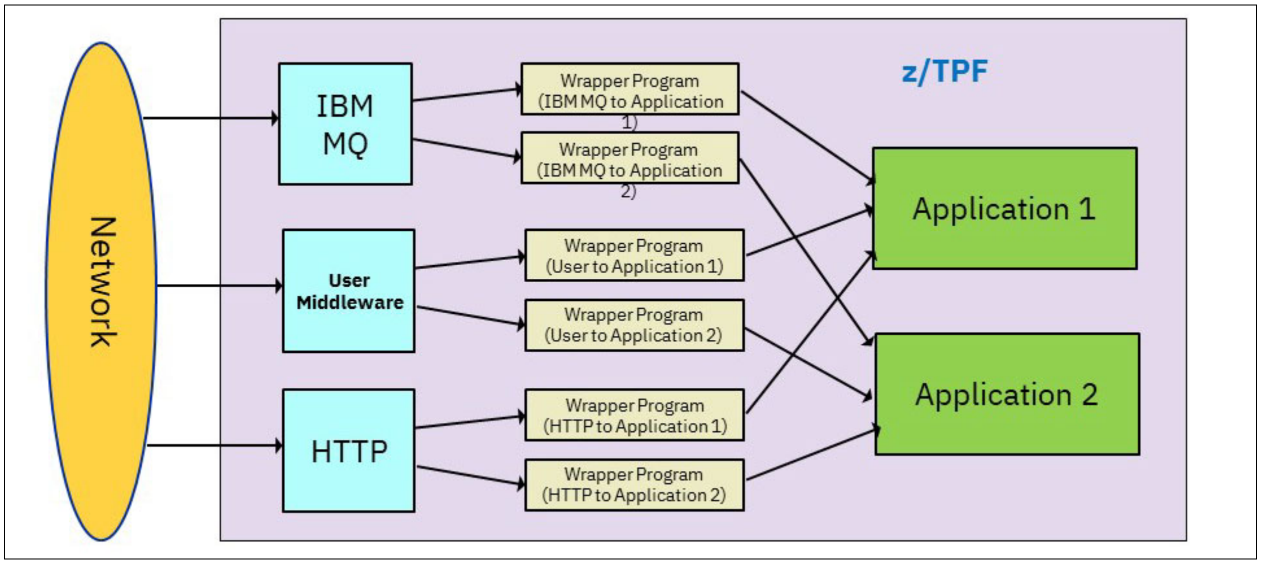

Network
IBM MQ
User Middleware
HTTP
Wrapper Program (IBM MQ to Application 1)
Wrapper Program (IBM MQ to Application 2)
Wrapper Program (User to Application 1)
Wrapper Program (User to Application 2)
Wrapper Program (HTTP to Application 1)
Wrapper Program (HTTP to Application 2)
z/TPF
Application 1
Application 2

Figure 20 Multiple communication protocols accessing applications

With the introduction of REST, you can quickly service-enable your existing applications without writing wrapper programs for data transformation. REST eliminates the need for wrapper programs by leveraging schema-based Data Format Description Language (DFDL) data transformation that does the message parsing, data transformation, and output message creation for you. Figure 21 illustrates how integrating DFDL into the REST stack provides data transformation for the application.

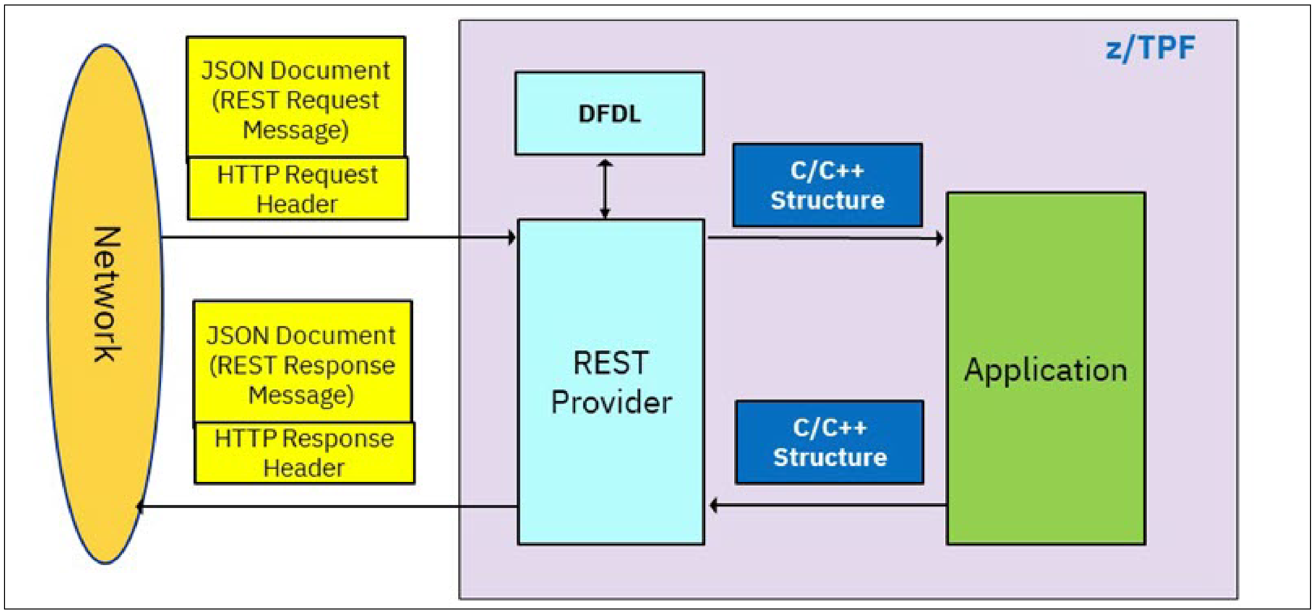


Figure 21 Using REST and DFDL to interface with existing z/TPF applications

IBM TPF Toolkit includes tools for developers to easily create REST services for the z/TPF system. The easy-to-use GUI walks a developer through the steps to create the artifacts that are needed to REST-enable applications on the z/TPF system. By using the REST Services View of TPF Toolkit, a developer can deploy the REST service to a z/TPF test system with a single click. The Rest Services View also provides an interface to the "Swagger UI" web-based tools to test your newly deployed REST service.

Figure 22 shows a case where you start with a monolithic z/TPF application that is accessed through a single network flow. As part of modernizing that application, you break it into many smaller microservices, each of which is its own API. By using this traditional microservices approach, you now have multiple network calls (API flows) in and out of z/TPF, which result in increased latency and excessive CPU overhead (on z/TPF and the service consumer platforms). A best practice to avoid those problems is to expose a single macroservice to the cloud application that gets you back to a single call into z/TPF. Then, an orchestration layer (ideally written in Java) calls the various microservices on z/TPF by using optimized local calls.

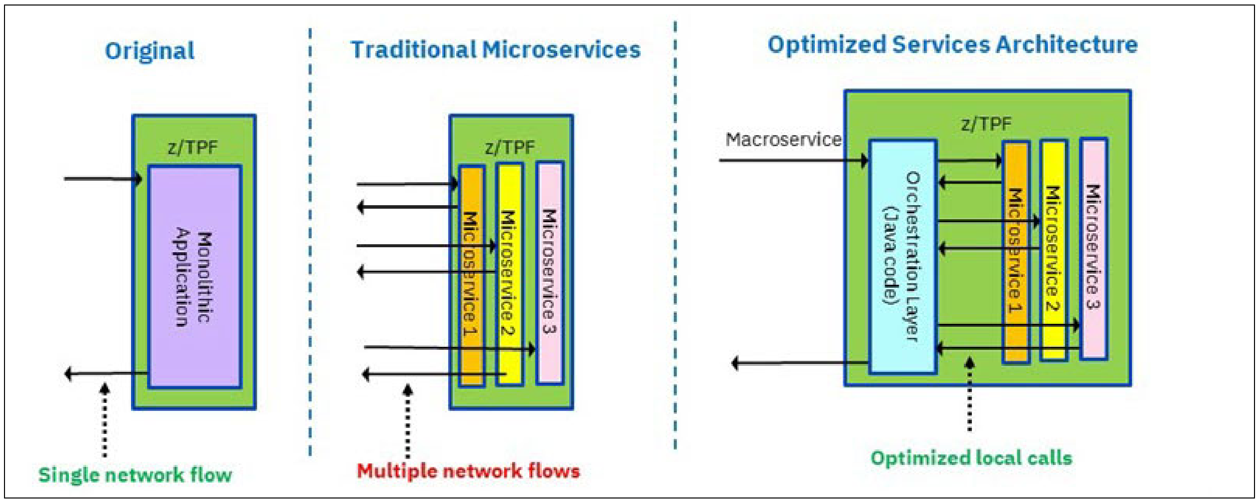


Figure 22 Orchestrating multiple service calls

z/TPF as a part of an event-driven architecture

Every organization has critical business processes that require real-time data to make decisions or update consumers and become more responsive. The different events that comprise a process must be efficiently communicated, acted on, and processed to derive insights and intelligence. This information must be shared in a fast, efficient, and flexible way that supports business processes that help make decisions or even report the right status to the consumer.

To achieve the real-time delivery of relevant information to business processes, many organizations are implementing event-based architectures and backbones across their enterprises. z/TPF applications form an inherent part of this event-based architecture because critical business data is derived from the applications and system of records.

Apache Kafka is often used for event-driven applications. Kafka-based event backbones such as IBM Event Streams or Confluent Platform can run in the cloud and on the mainframe on Linux on IBM Z.

z/TPF Business Events and Kafka are the primary methods that are used for sending events from z/TPF for the following reasons:

- When a z/TPF application detects a condition (like a sold-out airline flight) that is of interest to other systems in the enterprise, the application can send the message as a signal event or through Kafka to event consumers.
- For both signal events and Kafka, the z/TPF application program processing the transaction issues a function call that causes the event data to be placed on a local IBM MQ queue, and then control is returned to the application program to continue processing the transaction. Asynchronously, a separate system process pulls the message from the local IBM MQ queue and sends the event to the remote event consumers. Decoupling the event creation and event transmission ensures that event processing does not interfere or delay the processing of the real-time transactions. Staging the event data on a local IBM MQ queue allows remote nodes to consume event data at their own pace and protects the z/TPF system from impacts when z/TPF applications create events faster than remote consumers can read those events, which include cases where one or more remote consumers are down for a period of time.

Figure 23 shows z/TPF applications issuing signal events and the system asynchronously filtering, formatting, and transmitting those events to one or more event consumers.

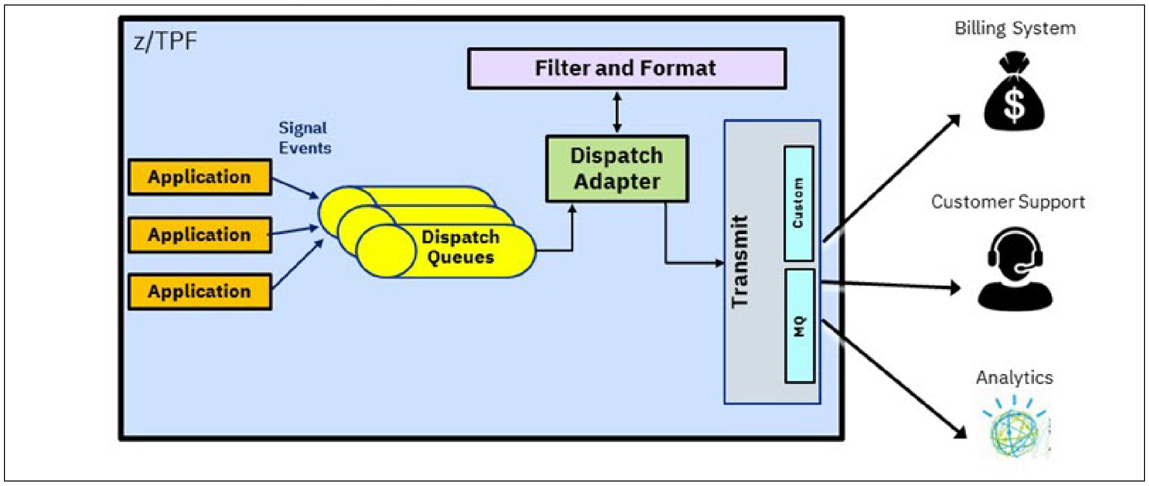


Figure 23 Architecture of z/TPF signal events

Figure 24 shows z/TPF applications generating Kafka events, with a Kafka producer asynchronously monitoring that queue and sending the Kafka events to a Kafka cluster.

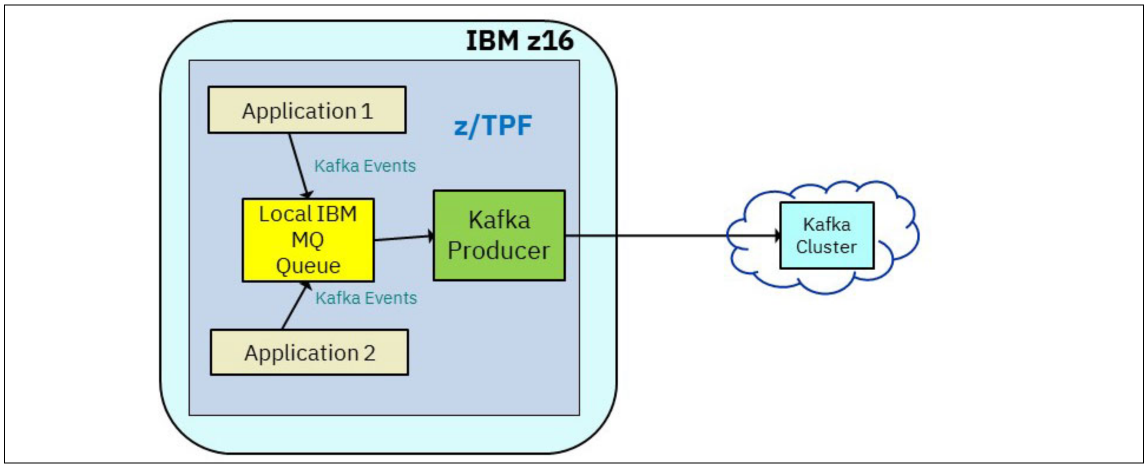


Figure 24 z/TPF integration with Kafka

Typically, the Kafka cluster is run outside of the IBM Z platform; however, IBM Event Streams can host the cluster on Linux on IBM Z under Red Hat OpenShift (in addition to cloud and distributed platforms). When the cluster runs on Linux on IBM Z, the adjacency of the cluster to z/TPF workloads can provide resilience and scale for event flows along with the improved security and governance capabilities that the IBM Z platform provides.

For a z/TPF event that must be sent to multiple remote consumers, by sending the event once from z/TPF to a colocated Kafka cluster on Linux instead of sending the same event multiple times from z/TPF, you can save CPU costs. The Kafka cluster architecture provides even more value if the event consumers are geographically distant. Distance-driven latency issues (or a remote consumer is offline frequently) can limit throughput. It is more economical to queue all that event data in Linux. By sending each event once from z/TPF, adding more event consumers in the future does not increase the event costs on z/TPF, as shown in Figure 25.

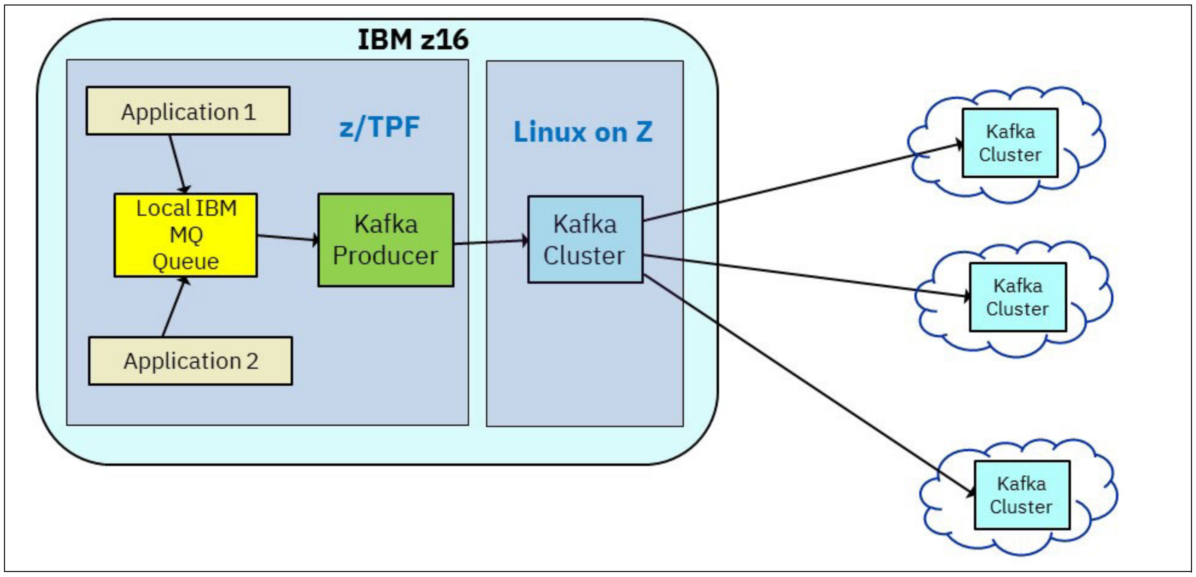


Figure 25 Using Linux on IBM Z to feed multiple Kafka clusters

After the events are in Kafka topics, they are available to be streamed across multiple cloud environments where the data can be used for notifications, analytics, or (projected, streamed, or used by) databases to provide local “in sync” copies of information. A few examples of using event processing through Kafka include getting real-time information for monitoring and analyzing trends; getting delivery notifications or processing notifications; or in business processes like ordering, claim handling, account debits, and credits.

IV. Simplifying information sharing and data access

Data is the foundation for all businesses and one of their most valuable assets. As a system of record (SOR), access to the data that is on the z/TPF system is critical for making business decisions in real time by leveraging analytics, AI, and machine learning (ML).

Depending on the business need, there are two aspects to accessing data on the z/TPF system:

- Real-time access to data to make better business decisions from the most current data, such as the balance for an account or the seat inventory on an airline.
- Near real-time access to replicated data to perform ad hoc queries, such as the transaction history for an account or flight schedules for a day, without impacting real-time transactions.

In both methods, the z/TPF system provides access to the data by using standard interfaces that do not require updates to z/TPF applications.

Accessing z/TPF data in real time with standard interfaces

In a hybrid cloud environment, systems of engagement (SOE) applications running on the cloud often must access, and potentially update, the latest copy of system of record data on z/TPF. For years, z/TPF data could be accessed remotely only by developing custom connectors, which required user code for both z/TPF and the remote systems. Writing and maintaining custom connectors was tedious, slow to deploy updates, and required custom governance for access control.

Because of the many difficulties that are associated with custom connectors, IBM now provides a way to access z/TPF data remotely by using a standard MongoDB client. With this capability, a client application can issue standard MongoDB APIs as though it is communicating with a MongoDB server, without any z/TPF knowledge. For example, you can use this support to access the latest copy of specific data, such as one specific airline reservation.

In reality, the MongoDB client is communicating with the MongoDB Interface for z/TPF layer on z/TPF and converts MongoDB wire protocols into the appropriate internal z/TPF database APIs. It uses DFDL to convert the data between z/TPFDF internal representation and the MongoDB format. There is zero user code that is required on z/TPF to accomplish this task. The z/TPF database administrator creates DFDL schemas for z/TPFDF data, and then they define which users are allowed to access which z/TPFDF databases. They also determine whether access is read-only or updates are allowed.

Figure 26 shows the same z/TPFDF databases that are shared (accessed) by remote applications by using MongoDB and local z/TPF applications that use z/TPFDF APIs. Therefore, remote applications get the benefits of the z/TPF database design and performance.

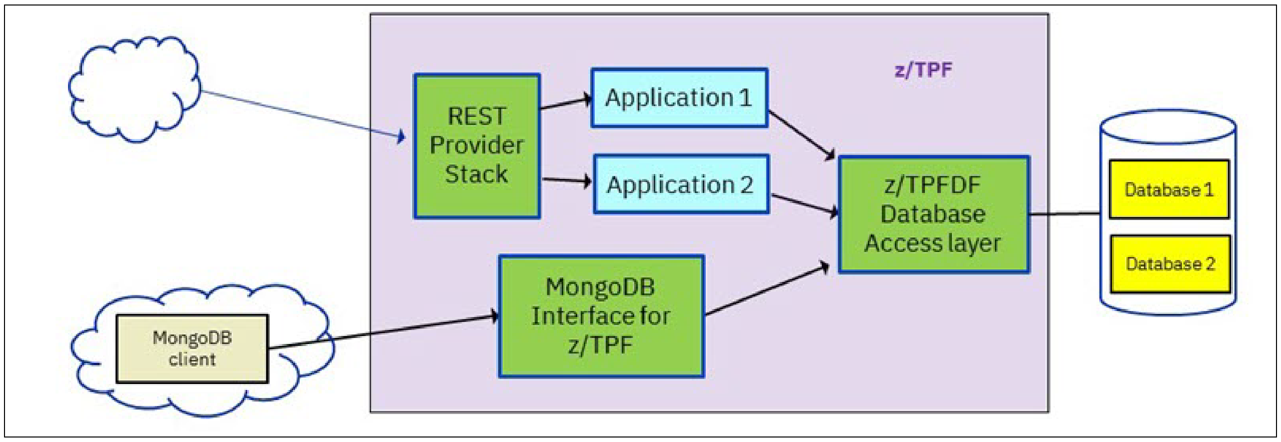


Figure 26 Local and remote access to the same z/TPF databases

If the remote client must access one specific record in one z/TPFDF database, the MongoDB Interface for z/TPF is a good fit. However, if the remote client must access multiple pieces of data that are in different z/TPFDF databases, exposing the combination of that data as a service (REST API) might be a better option. This situation is especially true if you want to centralize the logic (on z/TPF) that manipulates multiple databases and handles different versions of database records.

Optimized access to near-real-time data with events

There are other use cases where ad hoc queries must examine millions of database records to find all the entries that match a set of search criteria. You do not want these activities performed on z/TPF because they are resource-intensive and can interfere with real-time transactions. In an event-driven architecture, certain database record updates are events that trigger other processing, so a copy of that database update must be sent to one or more remote systems.

For years, it was the responsibility of the user (application programs) to propagate copies of database updates to the appropriate destinations. When an application program updated a z/TPF database, it placed a copy of the data onto a custom local queue, and then used a custom transport mechanism to send the data to the appropriate remote recipients. Most implementations sent the updates as a batch ranging from only once a day to multiple times per day, where there was delays of minutes to many hours in between the time the data was updated on z/TPF and the remote system received a copy of that data. Multiple z/TPF application programs had to be modified to make sure all places that updated a specific database were kept in sync, and code changes were also required whenever the list of remote recipients changed or the transport mechanism that was used for a remote recipient changed. Typically, raw binary data was sent that required each recipient to understand the format of the data, different versions of the data, and to convert the data to a format consumable by that remote data consumer. In other words, there were several difficulties with custom data offloads that were performed in the user code.

The z/TPF data events solution makes it easy to push copies of z/TPF data to remote data consumers and has the following advantages:

- No application program changes are necessary.
- Centralized control and configuration.
- Makes database updates available to remote consumers in real time (which also enables real-time business analytics).
- Automatically converts the data to a consumable format (XML or JSON) or custom format (leveraging DFDL).
- Filters the data such that each recipient gets only the data for which they have a business need (does not send sensitive data to a recipient that has no business need for that data).
- Augments the updated data with metadata to explain why the database record was updated. For example, a passenger decided to stay in town an extra day and pushed their return flight out one day versus the original return flight was canceled causing the passenger to be stuck in town one more night.
- Multiple transport mechanisms are available (IBM MQ, HTTP, Kafka, and others).
- Supports updates to z/TPFDF databases and older style FIND/FILE z/TPF databases.
- TE eligible.

The z/TPF data events solution allows the application to focus on the business logic to process transactions. By using a database definition, you control which databases, when updated, trigger the system to generate a data event, what remote systems get a copy of the data, how to filter and transform the data based on destination, and what transport to use to send the data event to that destination. No z/TPF application changes are required.

Figure 27 shows that when z/TPF applications update database records and the database definition says to generate a data event, the z/TPF system puts a message on a local IBM MQ queue, called a dispatch queue, that contains information about the data event, and then immediately returns control to the application program so that it can continue processing the transaction without delay. There is an asynchronous system background process that reads from that dispatch queue and for each remote destination, filters the data, formats the data event, and then transports it.

The decoupling of the code that updates a database record from the code that sends data events means that data events do not impact transaction response time, including cases where a remote data event consumer is not available or able to handle the volume of data events that are generated by z/TPF. In a steady state, if the remote node has enough capacity and there is enough network bandwidth between z/TPF and the remote node, then data flows in near real-time. However, if the remote node does not have enough capacity or there is not enough network bandwidth, then the data queues up in z/TPF and there are delays (seconds or even minutes) before the data arrives at the remote node. All the processing that formats and sends data events is TE-eligible.

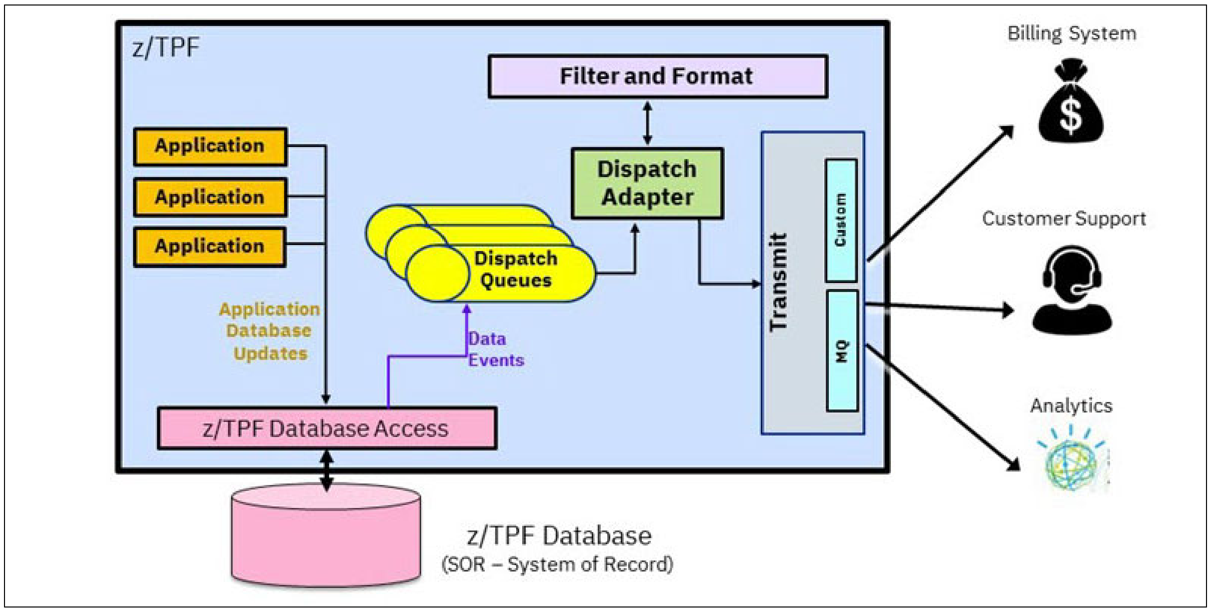


Figure 27 Architecture of z/TPF data events

Figure 28 on page 31 shows an example where the database record that was created or updated is a hotel reservation record that includes payment information like credit card number and expiration date, and personally identifiable information (PII data) like the guest's name, home address, and email address. A copy of the entire reservation record (PNR) is sent to the billing system because it has a business need for all the data. The payment information is filtered out and not sent to the customer support system because they have no business need for that data. All payment information and PII data is filtered from the copy of the data that is sent to the analytics system because that system has no business need for any of that data.

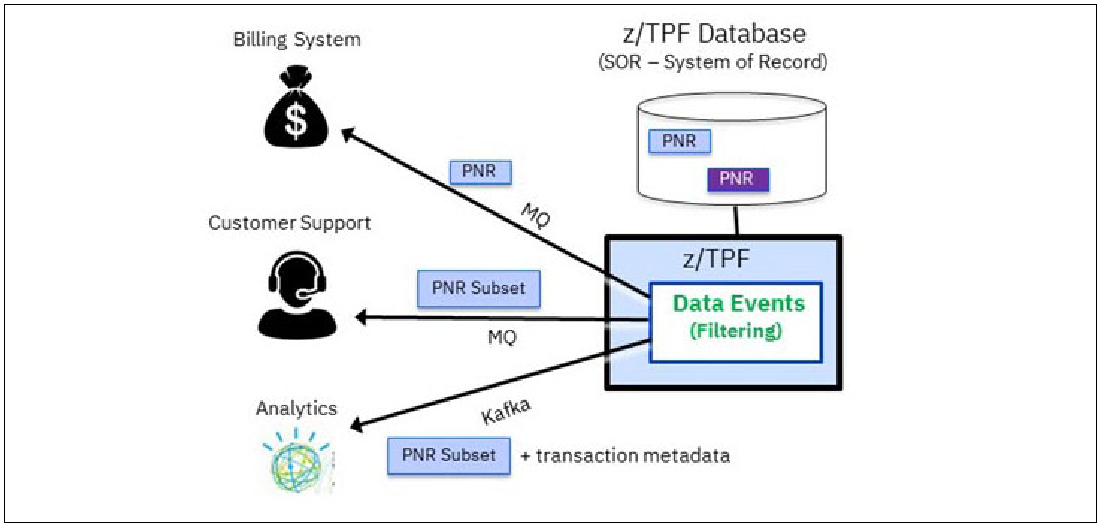


Figure 28 Filtering and feeding z/TPF data in real time

V. Getting more agile with enterprise DevOps

Enterprise DevOps is an important entry point for application modernization to increase business agility and productivity. With z/TPF and the IBM Z platform, you can leverage modern DevOps and a CI/CD pipeline.

Adopting a DevOps culture across your hybrid cloud enterprise, including IBM Z, helps to build a resilient software delivery process with the right security, governance, and quality gates in place. Leveraging DevOps practices in IBM Z development can potentially achieve higher business agility by reducing the time to value for creating business functions. By enhancing DevOps practices with automation and AI, organizations can focus on high-value initiatives and increase their competitive edge, such as improving customer satisfaction through faster, real-time problem identification and resolution.

The IBM vision and commitment to DevOps removes unnecessary differences in developing and managing applications on IBM Z so that the platform can easily integrate into your overall enterprise DevOps strategy. z/TPF is aligned with the IBM commitment to ensure enterprise standardization of tools and processes and require platform-specific capabilities only where necessary. Two key aspects of this strategy are as follows:

- Adopt or extend open-source and enterprise standard tools to z/TPF, including source code managers (SCMs), editors, pipeline orchestrators, automation for integration tests, and artifact repositories.
- Develop specific z/TPF capabilities in areas that should be platform-specific, such as unit test automation, building, and deployment of traditional z/TPF applications.

With this approach, z/TPF development and operations can have the same DevOps experience across the enterprise for on-premises or cloud, and in some aspects have an even better experience developing and managing z/TPF applications. DevOps can also play an integral role in providing solutions for the skills gaps on the platform because it allows you to attract new skills and accelerate the productivity of new developers and operators more easily.

Increased agility with DevOps and z/TPF

You can increase business agility and accelerate application modernization by adopting modern DevOps practices and integrating z/TPF into your CI/CD pipeline. For decades, z/TPF has supported continuous integration with the ability to nondisruptively deploy new applications on z/TPF production systems, and also has followed a continuous delivery model for providing system code enhancements and defect fixes.

Continuous integration and continuous delivery

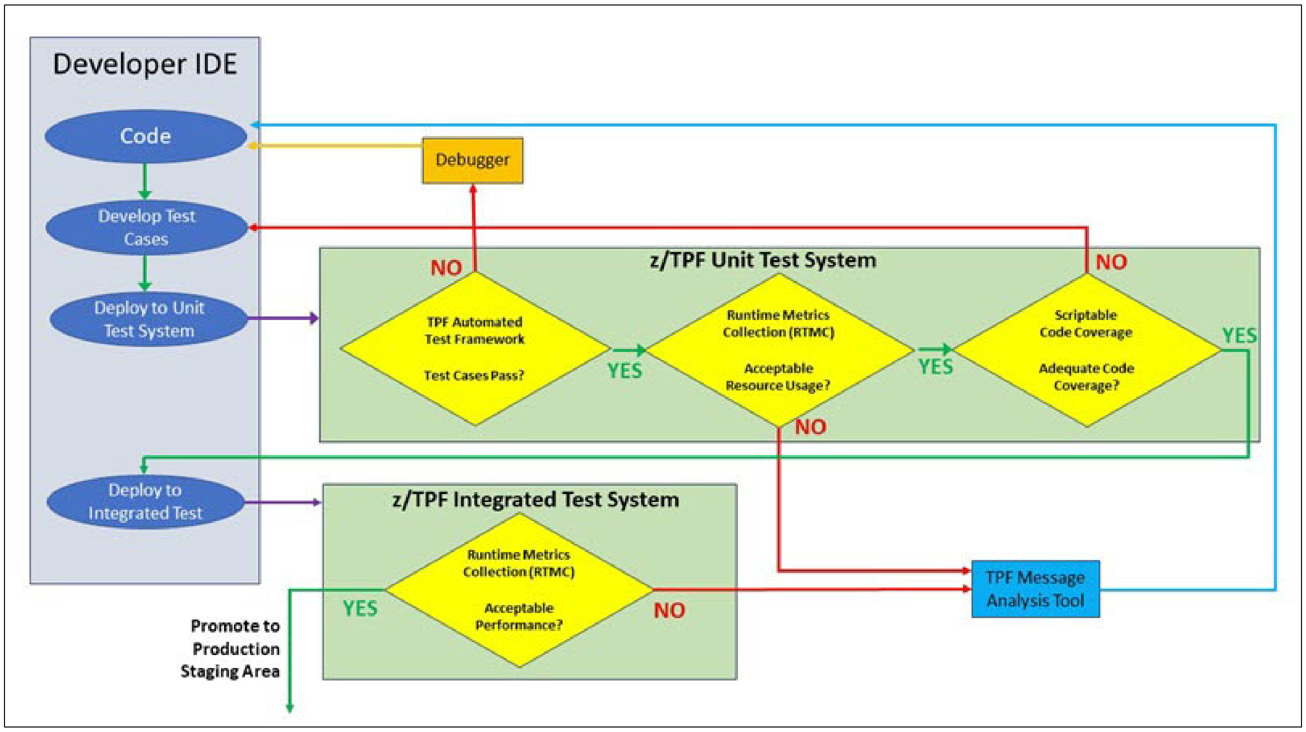
z/TPF is designed to support continuous integration and continuous delivery (CI/CD) business models. z/TPF is built for non-disruptive deployment of application updates to z/TPF production environments as often as needed, even multiple times per day, by using the z/TPF dynamic program loader. z/TPF is engineered so that application updates can be deployed with the z/TPF dynamic program loader while transactions are being processed, and new transactions automatically use the updated code without any scheduled downtime. In addition to seamlessly deploying new code without scheduling a system or application outage, the z/TPF dynamic program loader also allows you to back out problematic application changes.

To further increase business agility, z/TPF system code follows a continuous delivery model that allows you to accelerate deployment of new functions into production. z/TPF product enhancements and defect fixes are individually packaged and made available for deployment in customer environments, often weekly and at times daily. Whenever possible, system code updates are designed so that they can be deployed by using the z/TPF dynamic program loader, so fixes can be applied as soon as possible, and applications can take advantage of enhancements right away.

z/TPF development experience

From developing code through having the application ready to deploy in production, z/TPF provides an open development environment that can be incorporated into your overall enterprise DevOps strategy. Using tools such as the z/TPF Automated Test Framework, Runtime Metrics Collection, and the z/TPF Message Analysis Tool, you can leverage cost savings from shift-left issue identification and resolution, and overall faster time to value for new or modernized applications.

Figure 29 shows a recommended z/TPF DevOps workflow through the development cycle.



Developer IDE
Code
Develop Test Cases
Deploy to Unit Test System
Deploy to Integrated Test
Debugger
z/TPF Unit Test System
NO
TPF Automated Test Framework
Test Cases Pass?
YES
Runtime Metrics Collection (RTMC)
Acceptable Resource Usage?
YES
NO
NO
Scriptable Code Coverage
Adequate Code Coverage?
YES
z/TPF Integrated Test System
Runtime Metrics Collection (RTMC)
Acceptable Performance?
YES
NO
Promote to Production Staging Area
TPF Message Analysis Tool

Figure 29 z/TPF DevOps workflow

The three foundational elements of the development workflow are the integrated development environment (IDE), z/TPF unit test environment, and the z/TPF integrated test environment, all of which can be optimized to increase the value of DevOps for your business. For the IDE, enterprises can choose the best-fit solution for their development teams, either TPF Toolkit or a vendor IDE, to maximize productivity and possibly attract new talent. For both unit and integrated test environments, z/TPF automation and monitoring solutions can be leveraged to maximize shift-left savings and increase developer productivity.

Establishing an isolated unit test environment is important because it allows developers to perform predictable, repeatable tests and uncover problems early in the development cycle without impacting other developers. This environment is important when destructive testing or system environment changes are required. With z/TPF, developers can accelerate their testing and debugging in unit testing with key features, including the z/TPF automated test framework and code coverage. In addition, Runtime Metrics Collection can be used in unit testing to identify resource usage issues as early as possible in the development cycle.

An integrated test environment, typically a native z/TPF test system that can be shared by multiple users, provides the added value of testing on a larger scale, up to and including the size of and transaction volume of your production environment. Stress testing at scale is important when you are making significant application changes or adding major functions. Small application changes typically do not warrant full production scale testing. In the development workflow, new code is deployed to an integrated test system after unit testing is complete and the code coverage is acceptable. In this environment, the monitoring and observing tools, such as Runtime Metrics Collection, can be used to identify performance anomalies as a result of the new or changed application programs. When this testing is complete, the data can be used for capacity planning purposes to make sure that sufficient resources exist in the production environment before deploying the application changes into production.

The following sections provide more detailed approaches to optimize your z/TPF DevOps workflow across development and test by leveraging z/TPF functions and selecting the best-fit solutions for your environment.

Choosing an IDE

With z/TPF, enterprises can select the right IDE for their developers and extend the IDE capabilities by leveraging REST services that allow you to deploy code onto z/TPF test environments and enable code coverage on application code. In addition, with JUnit integration you can easily extend your IDE to run automated z/TPF unit test cases (Figure 30). The benefits of the IBM IDE solution, TPF Toolkit, are described in “Advantages of TPF Toolkit as your IDE” on page 37.

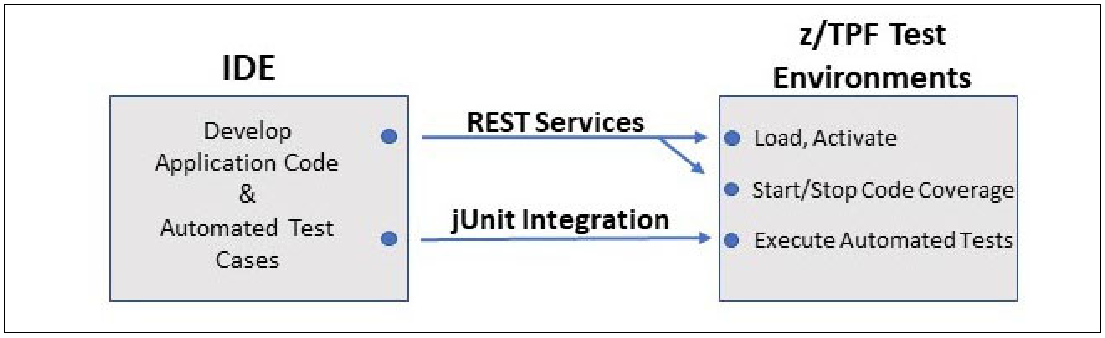


Figure 30 IDE integration with a z/TPF test environment

Maximizing DevOps shift-left savings in test environments

To leverage shift-left savings and increase agility, developers must have access to both isolated unit test and integrated test environments to identify and fix code issues as early as possible with the lowest business impact. By adopting a test strategy that leverages automation and monitoring solutions across all test environments, businesses can maximize developer productivity and business agility.

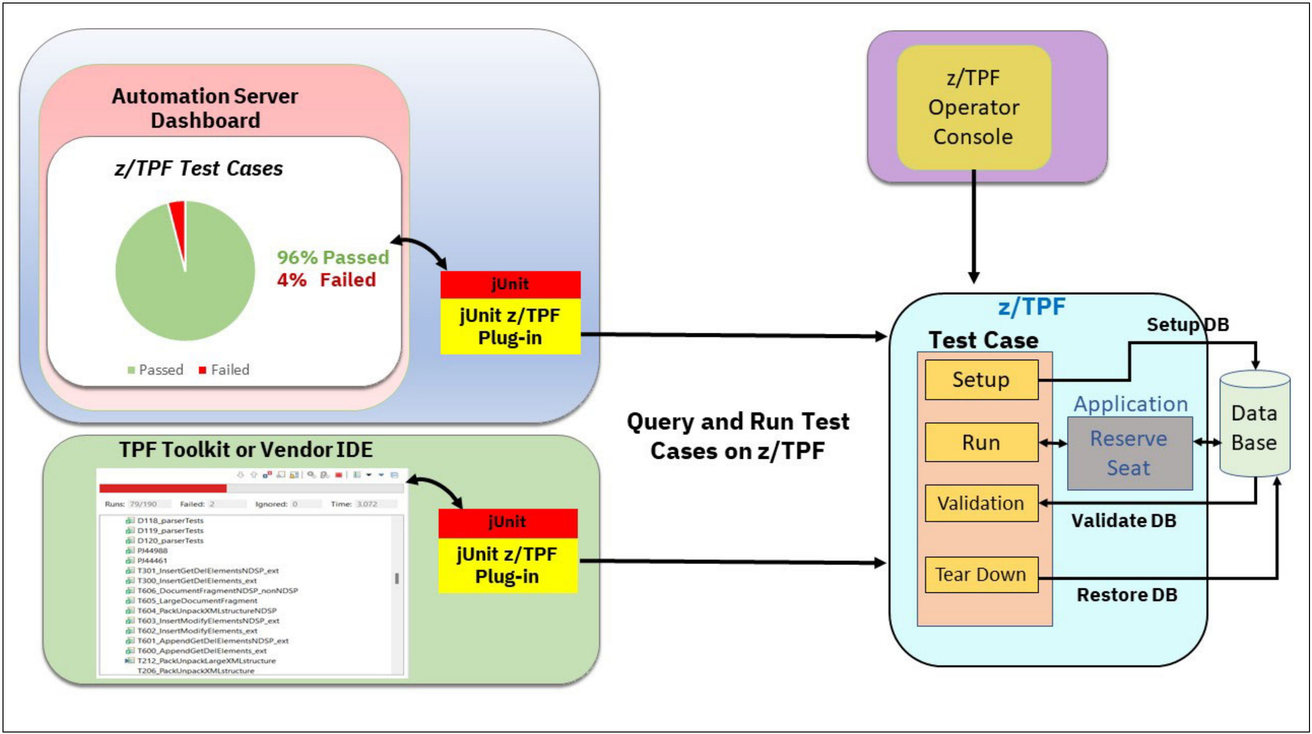
The z/TPF automated test framework

By leveraging automated testing, you can reduce the cost of testing, decrease the time to market, and improve the overall quality of code by testing earlier and continuously throughout the development cycle. z/TPF provides the z/TPF Automated Test Framework that is tailored for z/TPF applications that are written in any language from assembler language to Java while seamlessly integrating with your overall enterprise developer experience.

The z/TPF Automated Test Framework allows developers to create self-validating programmatic test cases, similar to other open testing frameworks. The framework is designed for agility and ease of use. For example, when new test cases are loaded in to the test system, they are automatically detected and able to run. When a developer is ready to test their code, they issue a z/TPF operator command or use a single click if they use TPF Toolkit to run the automated tests. The z/TPF Automated Test Framework also provides the option to override functions that the application may call in an attempt to drive various code paths within the application, allowing for more flexible and extensive automated testing. For example, if the z/TPF application calls a remote system but a test version of that remote server is not available in this particular test environment, by using the z/TPF Automated Test Framework, you can override that call to the remote system and return the results to your z/TPF application.

The z/TPF tests can be integrated into standard automation servers, like Jenkins, allowing continuous automated testing in integrated test environments. Using a JUnit plug-in, the z/TPF system can be integrated with virtually any well-known automation server.

Figure 31 shows the various ways of running tests and monitoring results against a z/TPF system.



```
Runs: 79/190    Failed: 2    Ignored: 0    Time: 3.072
D118_parserTests
D119_parserTests
D120_parserTests
PJ44988
PJ44461
T301_InsertGetDelElementsNDSP_ext
T300_InsertGetDelElements_ext
T606_DocumentFragmentNDSP_nonNDSP
T605_LargeDocumentFragment
T604_PackUnpackXMLstructureNDSP
T603_InsertModifyElementsNDSP_ext
T602_InsertModifyElements_ext
T601_AppendGetDelElementsNDSP_ext
T600_AppendGetDelElements_ext
T212_PackUnpackLargeXMLstructure
T206_PackUnpackXMLstructure
```

Figure 31   z/TPF automated test framework

Sometimes, full adoption of a z/TPF automated test strategy can take time due to the number of manual test cases that have grown over time. A progressive approach should be taken to focus on the fastest time to value opportunities first, similar to the mindset for application modernization that is described in "II. Enhancing and modernizing applications " on page 14. Building automated tests for application code that rarely has defects and has not been updated in many years should not be a high priority because that action does not provide much value. Instead, focus automated testing on high-value areas like code that is modified most frequently or historically has a high number of defects because that action provides the highest value and greatest return on investment. Also, automated test cases enable 'write once, test everywhere' value for applications because they can be used through the development cycle, from unit testing to integrated testing, to verify code changes.

Code coverage

When testing new application code, an important metric for the quality of testing is code coverage, which can be used to help determine whether an application is ready to deploy into production. Code coverage shows you which parts of your new and changed code ran during testing, allowing you to identify gaps in testing where you can create extra automated test cases to cover those gaps and increase test coverage. Using z/TPF Scriptable Code Coverage, developers can integrate code coverage support into test cases by using REST services to collect code coverage data for their application programs.

For example, a developer is running a test program for a new or changed application on z/TPF. To start code coverage, they send a REST request to z/TPF and then send a test input message to drive the z/TPF application. When the response is received, another REST request is sent to z/TPF to stop code coverage. This REST-enabled support allows z/TPF code coverage to be integrated into any testing platform that you choose, from vendor IDEs to open-source automation servers.

Monitoring in a test environment

Runtime Metrics Collection is a flexible and extensible monitoring and analytics framework that can be used to identify performance anomalies from a new or changed application in a test environment by collecting information such as CPU consumed, I/Os performed, and transaction response time. By leveraging Runtime Metrics Collection, issues with new and changed applications can be identified and fixed early in the development cycle, which reduces costs and time-to-resolution compared to discovering these problems in later-stage performance and capacity tests. For more information about Runtime Metrics Collection, see "Enhanced z/TPF monitoring with Runtime Metrics Collection and AI" on page 41.

When using Runtime Metrics Collection in a test environment, a developer can validate whether a new application is performing as expected by comparing transaction data from the new version of code against a baseline of the existing code on a Grafana dashboard.

Figure 32 shows a sample dashboard that displays Average CPU per transaction data for existing and new versions of code. The left part of the dashboard shows the CPU consumed per message by the existing application, and the right side shows the higher CPU consumed per message by the updated application. The developer can use this information to determine whether the new version is performing as expected, and whether the change in resource usage is acceptable.

For example, if you made a minor change to the application but Runtime Metrics Collection shows that the CPU consumed per message went up by 42%, you must investigate because the result is unexpected and likely unacceptable. However, if you added new function to your application that accesses and updates more databases, Runtime Metrics Collection showing an increase in average I/Os per message is expected.

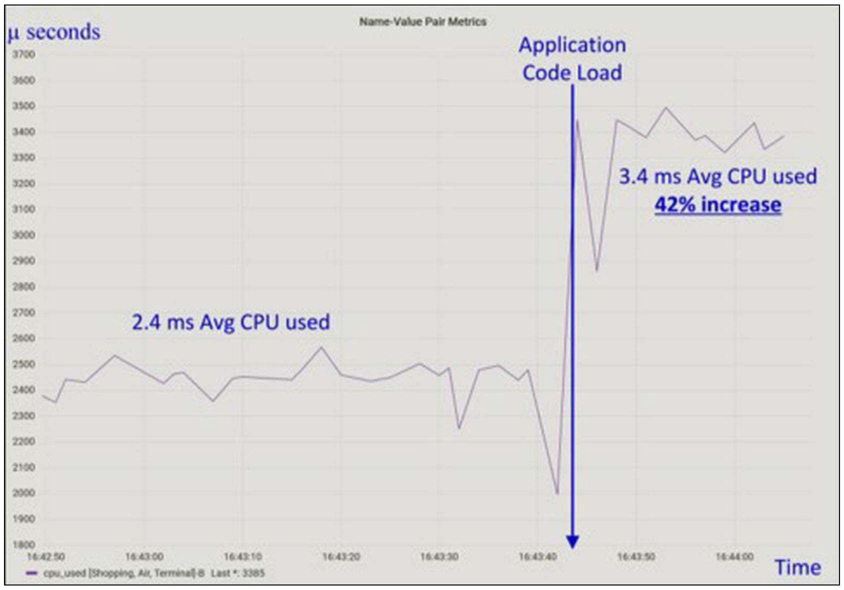
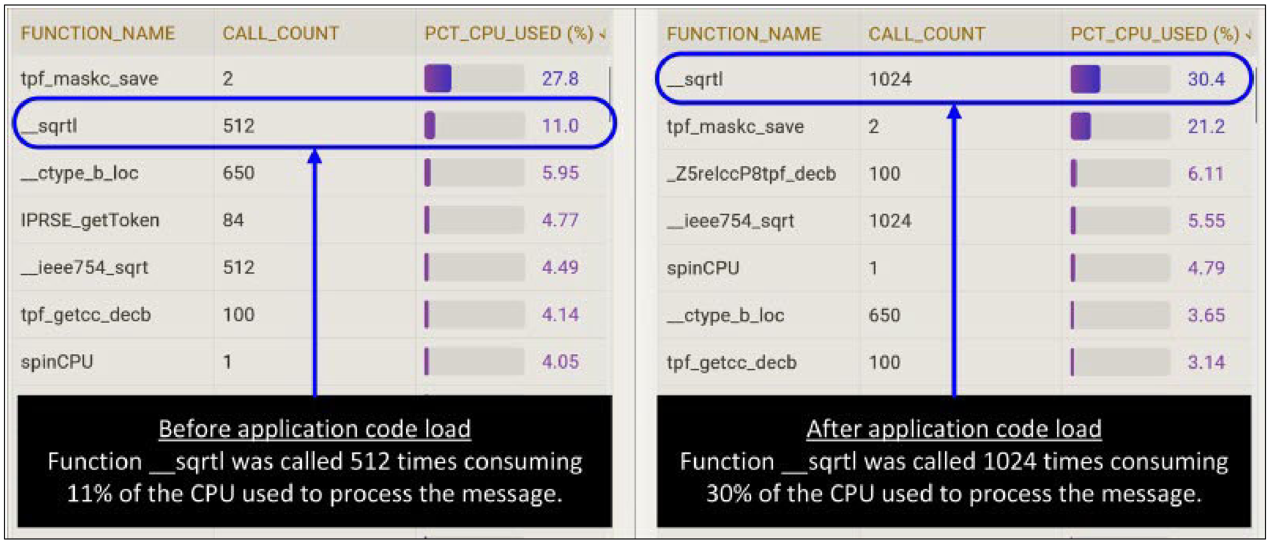


Figure 32 Runtime Metrics Collection dashboard showing the CPU impact of a program load

If the change in resource usage is acceptable, Runtime Metrics Collection data can be used by capacity planning to more accurately predict the impact of the new application version on the z/TPF production environment. By having early access to this information for analysis, the operations team has ample time to add resources to the production environment, if necessary, before deployment.

If Runtime Metrics Collection data identifies that resource usage by a new version of an application is unexpected or unacceptable, use the z/TPF Message Analysis Tool to trace and compare specific transactions for the application to identify the root cause.

Figure 33 on page 37 shows an example of a z/TPF Message Analysis Tool dashboard comparing transaction data for new and existing code. When the cause of the increase is identified, a business decision can be made to accept the performance or to iterate on the design to make it more efficient.



FUNCTION_NAME	CALL_COUNT	PCT_CPU_USED (%)
tpf_maskc_save	2	27.8
__sqrtl	512	11.0
__ctype_b_loc	650	5.95
IPRSE_getToken	84	4.77
__ieee754_sqrt	512	4.49
tpf_getcc_decb	100	4.14
spinCPU	1	4.05

Before application code load
Function __sqrtl was called 512 times consuming 11% of the CPU used to process the message.

FUNCTION_NAME	CALL_COUNT	PCT_CPU_USED (%)
__sqrtl	1024	30.4
tpf_maskc_save	2	21.2
_Z5relccP8tpf_decb	100	6.11
__ieee754_sqrt	1024	5.55
spinCPU	1	4.79
__ctype_b_loc	650	3.65
tpf_getcc_decb	100	3.14

After application code load
Function __sqrtl was called 1024 times consuming 30% of the CPU used to process the message.

Figure 33 z/TPF Message Analysis Tool dashboard

The z/TPF Message Analysis Tool is built into the Runtime Metrics Collection analytics pipeline by leveraging the same open-source components, such as Grafana.

Solutions to optimize your DevOps experience

In addition to following a DevOps workflow, the developer experience can be optimized by choosing the best-fit products to meet the needs of your development organization. Two products that can potentially increase agility and productivity are TPF Toolkit as the IDE solution and Virtual PARS (VPARS) as a unit test solution.

Advantages of TPF Toolkit as your IDE

When using TPF Toolkit 4.6 (an Eclipse based solution) as your IDE, you can significantly enhance your DevOps experience for assembler language, C/C++, and Java development. In addition to creating or updating existing applications, TPF Toolkit 4.6 provides many features that are optimized for the z/TPF build environment, including seamless integration with an automated test framework and code coverage tools, an integrated debugger, and the ability to deploy code to a z/TPF test system with a single click. TPF Toolkit also provides numerous wizards that can increase the agility of application modernization, such as creating REST services (see “III. Integrating across hybrid cloud” on page 22) and offering z/TPF specific views to provide an overall better user experience.

Figure 34 shows the features of TPF Toolkit. A developer codes the applications and creates automated test cases within the TPF Toolkit editor. Then, by using a clickable action, the entire project can be built. After the project is built, the developer can click a button to load, activate, and test the interfaces with the z/TPF dynamic program loader, and then invoke the automated test framework to run the test cases. If there are problems with any application code changes, an integrated z/TPF debugger can be used to step through the code to diagnose the problem. As part of the testing, a developer can optionally enable code coverage for application code to identify how much of the application code was tested when driving the automated tests. Tools that a developer needs to develop code on the z/TPF platform is integrated into one solution.

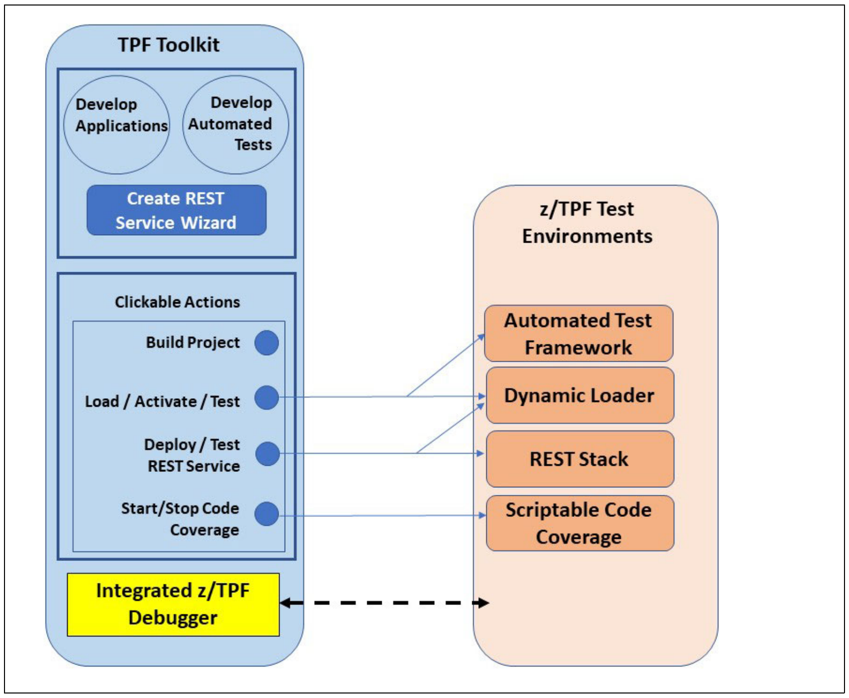


Figure 34 TPF Toolkit integrated with z/TPF test environments

A key aspect of enterprise DevOps and application modernization is the automation of testing. One advantage of choosing TPF Toolkit as your IDE is that it is tightly integrated with the z/TPF Automated Test Framework, which provides a path for organizations to grow their automated test inventory. With TPF Toolkit, a developer can create applications and corresponding automated test cases and load them with a single click of a button.

By adopting a practice of co-creating applications and automated test cases, you can immediately realize the benefits of automation for your development teams and progressively grow toward your organization's overall automation goals.

TPF Toolkit also features integrated debugging facilities that allow developers to step through code while it is running on the z/TPF test system, increasing developer agility. The z/TPF debugger, which is designed for C/C++ or assembler language developers, is feature-rich with capabilities such as displaying stack variables, displaying memory, and setting breakpoints. When a problem within the code is identified, debug sessions allow developers to step through the code instruction by instruction to more quickly determine the root cause. Because TPF Toolkit is built on the Eclipse framework, Java developers can leverage the standard Eclipse Java debugger to identify and fix bugs in Java code running on z/TPF. Because this solution is open standard, there is little to no learning curve for Java developers to learn how to debug Java code on z/TPF.

TPF Toolkit can further increase agility and add value to your DevOps model with code coverage tools, which can help determine whether an application has been adequately tested before deploying to production. Using the TPF Toolkit code coverage feature, a developer can see how much of their application was run by all the test cases, and which specific source code lines were not tested. This information can be used to determine which, if any, extra test cases are required and can help prevent problematic applications from being deployed into production. For updates to existing applications, the TPF Toolkit code coverage results can be output into a format consumable by SonarQube to take advantage of analysis of only new and changed lines of code in the application.

VPARS for isolated unit test systems

One potential challenge when adopting a DevOps model with agile development is establishing an effective, cost-optimized unit test environment. A common solution is to use the Virtual PARS (VPARS), a Virtual Software Systems, Inc. offering, which allows multiple virtual machines to transparently share a z/TPF database by using IBM z/VM®. This solution provides developers with isolated z/TPF test environments so that you can enable testing across your agile teams and support your CI/CD pipeline. VPARS test environments can even be provisioned for specific projects or for use in nightly automated test runs. Because VPARS runs on the IBM Z platform, you are testing on the same hardware as your production environment (not a simulator) and can leverage the sharing of hardware and CPU to optimize your TCO (for more information, see “Maximizing CPU resources ” on page 9).

VPARS allows you to create many isolated z/TPF test systems without each one needing a full copy of the z/TPF database. For example, Figure 35 shows multiple z/TPF test systems within an IBM z/VM LPAR.

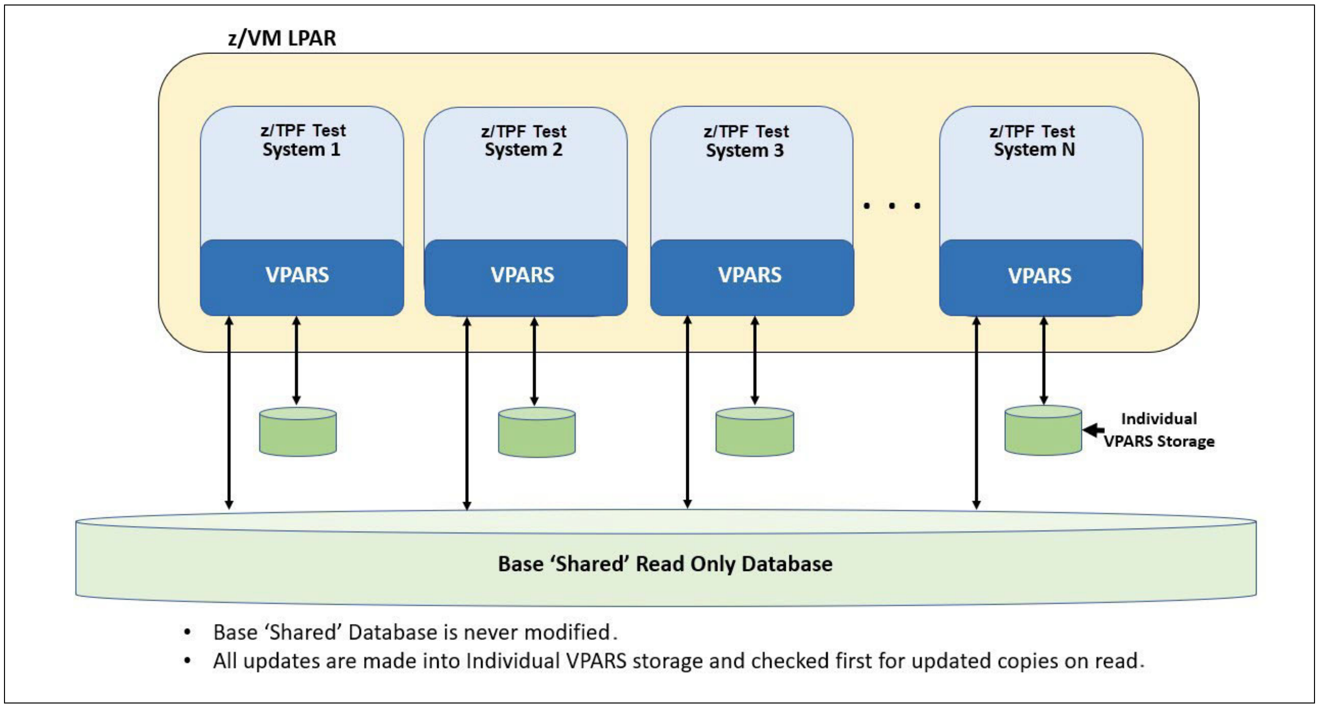

z/VM LPAR
z/TPF Test System 1
z/TPF Test System 2
z/TPF Test System 3
z/TPF Test System N
VPARS
VPARS
VPARS
VPARS
Individual VPARS Storage
Base ‘Shared’ Read Only Database
• Base ‘Shared’ Database is never modified.
• All updates are made into Individual VPARS storage and checked first for updated copies on read.

Figure 35 Multiple z/TPF test systems that are using VPARS

Each z/TPF test system has a VPARS layer that has access to the one full copy of the z/TPF database that is read-only, and to its own smaller individual VPARS storage area that is used to store any database updates made by that z/TPF test system. In this example, the full z/TPF database is 1 TB. To have 200 test systems each with a complete copy of the database, you need 200 TB of storage. However, with VPARS, each test system has a smaller storage area, which is 50 MB in this example, which is used to hold all database updates that are made by that test system.

When reading a record, VPARS first checks whether this test system already has an updated copy in the individual VPARS storage, and if not, VPARS gets the record from the base shared read-only copy of the database. In this example, 200 test systems without VPARS require 200 TB of storage (1 TB for each test system), but the test systems with VPARS require only 1.01 TB of storage (1 TB for the one shared full copy of the database and 10 GB total for 200 instances of private 50 MB writable storage for each test system), which is a more than 99% reduction in unit test system storage requirements.

z/TPF operations experience for leveraging AI insights

In addition to the developer experience, z/TPF enables enterprise DevOps practices for operations as a seamless extension to your CI/CD environment. The z/TPF monitoring solutions can help you identify and resolve issues faster and more efficiently, both at a system level and as part of your broader enterprise.

For many years, the primary monitoring and diagnostic tools that were used by z/TPF operations have included the following tools:

- The z/TPF data collection and reduction tool captures system data for a single z/TPF image for a period of time, which includes system-wide summaries for input messages, system resource usage, application programs, and databases. Data collection typically runs 1 - 2 times per day during peak traffic periods to gather information to validate that the system is performing as expected and configured properly to handle your workloads.
- Continuous data collection (CDC) collects certain system data so that you can do real-time monitoring of your z/TPF system.
- The z/TPF software profiler is a real-time sampling tool to help you identify code hot spots and other resource usage issues. The tool can be run on a production z/TPF system with near-zero overhead. As a best practice, run this tool periodically to identify performance issues (and then fix them) before they impact your workload.
- Client-written tools and procedures exist that heavily rely on SMEs to interpret the results based on their decades of experience. An example might be screen-scraping console logs for periodic application counter messages that are processed at the end of the day to generate reports.
- Various system monitors to alert you if any of the following situations occur:
  - Many application instances are queued and waiting for the same database lock.
  - A large queue is growing for one of more DASD devices.
  - Any application instance is using more resources than what you define as acceptable.
  - A network connection appears to be stalled.

z/TPF now also provides the following tools to leverage AI for deep insights without relying on SMEs with decades of experience to interpret the data:

- Runtime Metrics Collection provides real-time metrics, AI-infused insights, and visibility, which can be used with ML to further accelerate problem detection and resolution and maximize system resources. By leveraging ML and historical data to build models for what normal operations look like for your z/TPF infrastructure, the z/TPF operation teams can get advanced warning of anomalous activity that might lead to an SLA-impacting event. By putting the work on the machine to automate threshold setting for alerts, these organizations can reduce their reliance on SMEs to set and manage thresholds in their traditional monitoring solutions. These results and insights can be fed into enterprise monitors or trigger automated responses on z/TPF through IBM TPF Operations Server.
- The z/TPF Message Analysis Tool can be used in production to get an in-depth view of a what a specific instance of a transaction is doing. You take the insights that are gleaned by Runtime Metrics Collection to dig into the application for that type of transaction to understand exactly where the resources are being used.
- TPF Operations Server provides an interactive console to control the operation of z/TPF. It also provides the ability to capture and process console logs and automate system administrative tasks. TPF Operations Server console logs can also be fed into analytics servers to recognize patterns of z/TPF messages and automation can be leveraged to automate responses to situations on the z/TPF system.

Enhanced z/TPF monitoring with Runtime Metrics Collection and AI

Runtime Metrics Collection is a flexible and extensible monitoring and analytics framework that is built for monitoring the z/TPF operating system. Runtime Metrics Collection is built on open-source technologies that include Docker, MariaDB or MySQL database, Apache Kafka, Python, and Grafana. The framework is fully customizable so that organizations can add custom metrics, analytics, dashboards, advanced AI and ML, and other monitoring and analytics components. Because of its flexibility and extensive capabilities, Runtime Metrics Collection is the monitoring and analytics solution for both z/TPF test and production environments, and it can support enterprise-level observability.

The flexibility of this framework allows you to integrate your operational, application, business, and other metrics into your enterprise monitoring and analytics solutions. This tool allows you to monitor at system, transaction, application, and business levels so that you can optimize performance and business outcomes.

Runtime Metrics Collection allows you to monitor z/TPF in real-time at the following levels:

- System level: CDC provides high-level metrics for CPU, memory blocks, disk, TCP/IP, database, and other resource usage. You can also monitor Java on z/TPF and the resources that are used by all JVMs. Dashboards exist that allow you to view and compare metrics across all processors in a loosely coupled complex and see how these metrics have changed over time. These metrics allow you to see how issues are manifesting themselves at the system level.
- Transaction level: Name-value pairs allow you to annotate your transactions in many dimensions, such as message type, country of origin, and business partner. Name-value pair collection provides a set of dashboards that allow you to see the resources that are used by a type of message, a type of message from a country of origin, and so on, which allows you to understand the resources that are used at a high-level categorization, such as country of origin, or to find a particular message type that is sent by a particular business partner is having resource usage issues due to the transaction input parameters. This approach allows you to go from seeing issues at the system level down to the transactions that are causing the issue, and to identify issues that are caused by transactions, even though the overall system appears healthy.
- Application level: Using a program configuration file, you identify code packages or phases of processing for transactions without any application code changes. The name-value pair collection dashboards allow you to see the resources that are used by a code package or a combination of a code package and transaction-level name-value pairs. For example, a message type that is sent by a particular business partner is having resource usage issues due to the transaction input parameters in the fraud code package. You go from seeing issues at the system and transaction level down to the application code that is manifesting the issue. This application-level monitoring also includes the specifics of the operation of your Java application running on z/TPF and enables you to see the health of your application leveraging the JMX application data.

The solution can also be extended, so you can introduce custom metrics for the operation of your z/TPF system. You can implement metrics at the system level, such as a counter of how many calls were made to a business partner and whether timeouts are occurring. You can implement metrics at the transaction or application level, such as a counter of how many calls were succeeding or failing. You can implement metrics at the business level, such as whether a promotional sale is working or seats are selling quickly (and prices can be raised). In this way, you can monitor the specifics of your z/TPF applications to optimize diagnostics, business results, and more, all in real time.

Because Runtime Metrics Collection provides all these varying metrics in a single framework, you can implement advanced analytics, alerts, and dashboards that synthesize insights from the various monitoring levels. Within this framework, the data is captured in real time and historical data is retained in a database of record. You can build ML models to represent what normal looks like on your z/TPF system and evaluate models in real time to identify deviations from normal or trends of interest. These insights can be presented on dashboards or proactively generate alerts of any kind to signal operational or business SMEs to respond to situations of interest. These insights can also be fed to enterprise monitors (such as application performance monitors) to raise the visibility of z/TPF in your enterprise. Further, you can implement advanced AI decision logic to predict and respond to situations before your customers are adversely affected.

Figure 36 shows the overall architecture of the Runtime Metrics Collection solution.

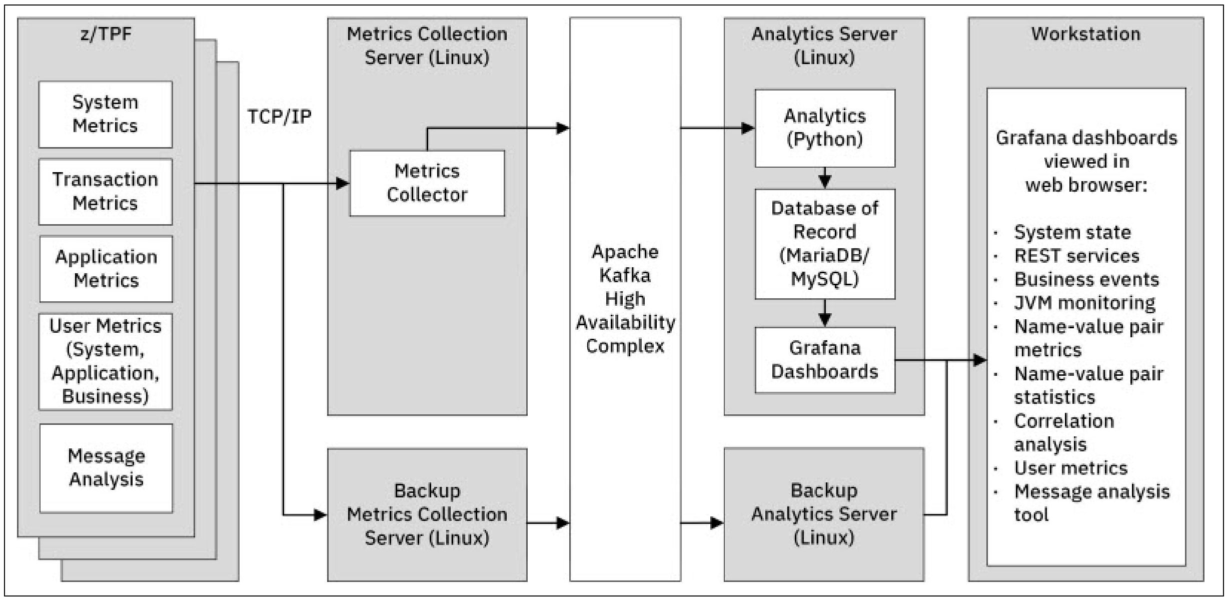


Figure 36 Runtime Metrics Collection architecture

Runtime Metrics Collection: A diagnostic use case that is using ML and AI

Figure 37 shows the point when the actual CPU usage (actual_cpu) of your z/TPF system deviates from the normal usage pattern (expected_cpu). Runtime Metrics Collection provides a name-value pair collection-based ML algorithm that calculates what the overall system CPU usage should be based on the current message mix, message rates, and what the predicted average CPU used is by each message type. Simple threshold detection or advanced AI can be put into place to generate alerts for deviations of the actual CPU from the expected CPU. Further, advanced models can be built that are based on the day of the week or month, seasonality changes, and so on, to detect deviations from normal while not raising false alarms for code loads, airline re-accommodation events, sales events, and more.

In this example, the actual CPU utilization on z/TPF was matching the expected CPU utilization that was calculated by Runtime Metrics Collection, but then suddenly the actual CPU utilization jumped and stayed higher than the expected CPU utilization. What caused this utilization increase?

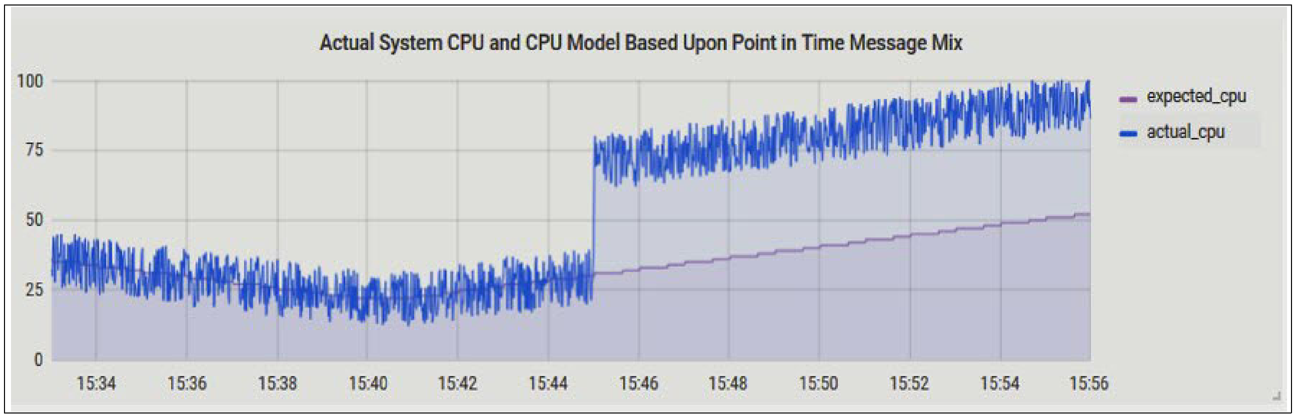
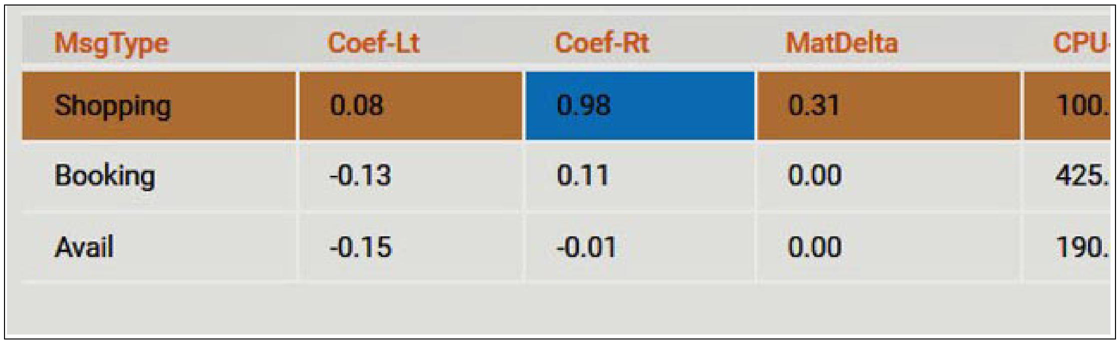


Figure 37 Runtime Metrics Collection detects an unexpected rise in CPU utilization

Runtime Metrics Collection includes sample, AI-based correlation analysis to determine whether transaction, application, or combinations of these metrics are correlated to system-level metrics. In this example, Figure 38 shows that the average CPU used by Shopping messages correlates to the rise in overall CPU usage on the system, which indicates that individual Shopping messages are using more CPU because the overall CPU usage of the z/TPF system is rising. This type of data is important because it helps target problematic applications and rule out things. For example, a recent change to the Booking application might have occurred, but from this data we can deduce that the CPU utilization increase was not associated with specific program loads or any changes on the z/TPF system. Before Runtime Metrics Collection, it might have been assumed that the recent change to the Booking application was causing the problem, which resulted in wasted time investigating that path and allowing the problem to continue for a longer period of time.

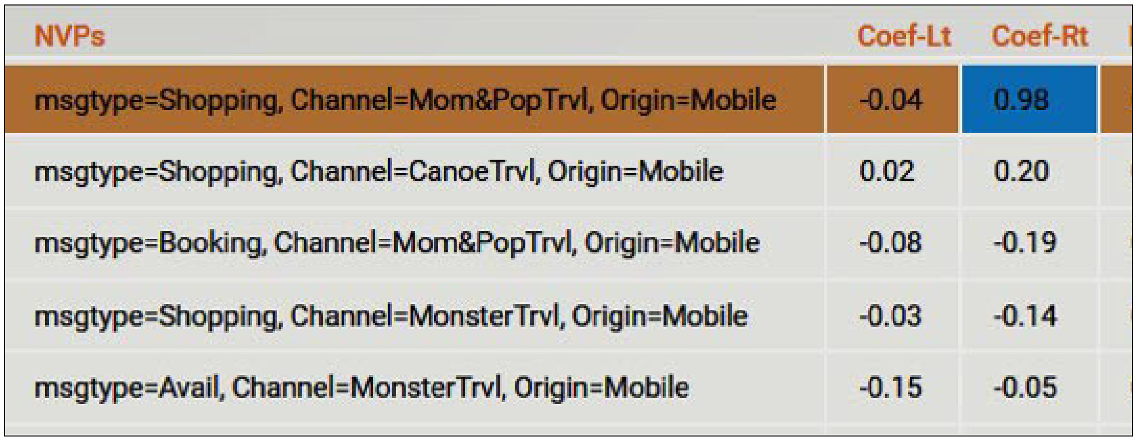


MsgType	Coef-Lt	Coef-Rt	MatDelta	CPU
Shopping	0.08	0.98	0.31	100.
Booking	-0.13	0.11	0.00	425.
Avail	-0.15	-0.01	0.00	190.

Figure 38 *Runtime Metrics Collection AI indicates that Shopping traffic is causing high CPU utilization*

The Shopping application was not updated recently, but the increase in CPU utilization is associated with the Shopping messages. The next step is figuring out whether the problem is the Shopping application itself or one or a subset of users of the Shopping application. To accomplish this task, use the AI analysis that Runtime Metrics Collection provides to dive deeper into the analysis of the Shopping messages by user, origin, and so on.

Figure 39 shows that only one user (Mom & Pop Travel) is generating Shopping messages, which are highly correlated to the rise in CPU utilization.



NVPs	Coef-Lt	Coef-Rt
msgtype=Shopping, Channel=Mom&PopTrvl, Origin=Mobile	-0.04	0.98
msgtype=Shopping, Channel=CanoeTrvl, Origin=Mobile	0.02	0.20
msgtype=Booking, Channel=Mom&PopTrvl, Origin=Mobile	-0.08	-0.19
msgtype=Shopping, Channel=MonsterTrvl, Origin=Mobile	-0.03	-0.14
msgtype=Avail, Channel=MonsterTrvl, Origin=Mobile	-0.15	-0.05

Figure 39 *Runtime Metrics Collection AI indicates that one specific user of the Shopping application is causing high CPU utilization*

You also can use AI correlation analysis to see that almost all Shopping messages from Mom & Pop Travel are specifying the full city search input option, which is the cause of higher CPU utilization for those messages, and because of the volume of those messages, Mom and Pop Travel also are the cause for higher CPU utilization on the system.

Using ML and AI correlation analysis on Runtime Metrics Collection, you identified the cause of the problem in 1 - 2 minutes. Using this analysis, you contact Mom & Pop Travel, and in this example, they realized they accidentally changed the Shopping default search option to perform a full city search. They correct the issue, and z/TPF CPU utilization returns to normal.

Runtime Metrics Collection: Enterprise monitoring

Over the decades, the processing of transactions has grown in complexity. Transactions that were processed by z/TPF in the 1960s were sent from travel agent and airline consoles directly to z/TPF by hardwired connections. Today, a multitude of devices on public (untrusted) networks send requests to components that are hosted in the cloud that connect to secure networks that route pieces of a request to server farms, z/TPF, business partners, and more entities in a multi-stage processing flow. In today's environment, it is critical to have enterprise observability to quickly identify which system is violating SLAs so that these issues can be resolved as quickly as possible.

You can extend Runtime Metrics Collection to stream data to an Application Performance Monitor (APM) such as IBM Instana®, Splunk, DynaTrace, or other vendor products by leveraging the OpenTelemetry observability framework. APMs allow your enterprise system reliability engineer (SRE) to monitor and apply analytics to be notified about when a situation of interest occurs, which systems and transactions are affected, which system is likely to have the source of the problem, and more. Then, the SRE can leverage the additional metrics and analytics in the APM for that system to gain a high-level understanding of the issue. Because z/TPF metrics can be included in enterprise APM solutions, z/TPF is not a mysterious black box, but can be seen like any other system in your hybrid cloud enterprise.

Figure 40 illustrates a transactional workload where multiple systems are involved in processing a transaction. Without an APM framework in place, your SRE can monitor the response times for user requests at the entry point of your enterprise at system A. However, if response times violate SLAs or transactions are ending in error, a “fire drill” ensues where the SMEs for each system play the blame game and attempt to prove that the cause is not on their system. With an APM in place, your SRE can see the response times, error rates, system-level metrics, and lower-level metrics for individual transactions for each system that is involved in processing the transactions on centralized web-based dashboards. APMs like IBM Instana include advanced AI analytics and alerting to identify the types of transactions that are affected by an issue, on which system the problem originates, and how the other systems are affected. By integrating z/TPF into your enterprise APM, you can significantly reduce the time that is required for problem determination and resolution, which limits your exposure to violating SLAs.

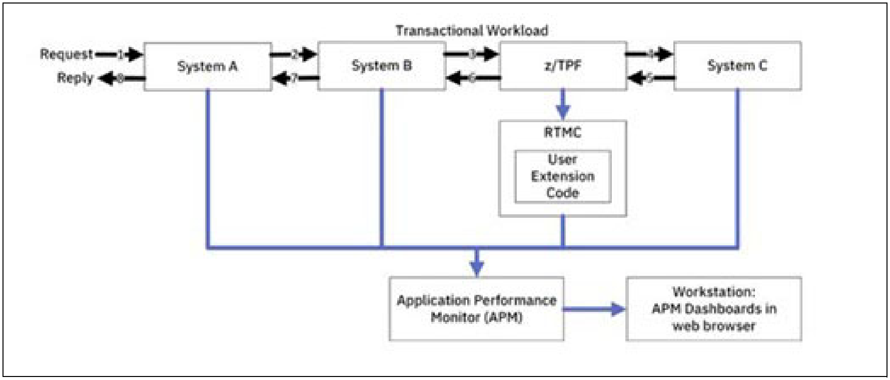


Figure 40 Using an APM to monitor a multi-platform workload

If the SRE that uses the APM dashboards determines that the z/TPF platform is the cause of the problem and more information is required to diagnose or rectify the issue, their investigation can move from using the enterprise APM to z/TPF tools like Runtime Metrics Collection, which provide more in-depth information than is available at the APM level.

Runtime Metrics Collection: Automating system response

Runtime Metrics Collection can be configured to retain historical data that can be used to train AI and ML models to predict issues and automatically take proactive steps instead of having a human react after an event has occurred.

Figure 41 shows a sample architecture that is used to automatically respond to various situations. Suppose that a specific business partner is submitting requests at an unusually high rate, which is increasing utilization on z/TPF. Based on business rules, historical metrics, and business and system metrics that are captured by Runtime Metrics Collection in real time, your AI analytics can evaluate, predict likely outcomes, and automatically respond to this situation for optimal business outcomes. If the increased rate of requests is generating more revenue, the analytics can trigger TPF Operations Server to run a script to add CPU capacity to the z/TPF system by using the Dynamic CPU feature to ensure that the system can be leveraged to capture that extra revenue. If the increased rate of requests is performing busy work that is not generating revenue, the analytics can trigger the APIM server to restrict the incoming transactions for that specific business partner to ensure that CPU capacity is reserved for revenue-generating requests. You can apply this same methodology to various business and operational situations by using AI frameworks that leverage the hardware-accelerated AI capacity of the AIU on the IBM z16 server.

The implementation of this framework can be staged as you develop your AI analytics and decision framework. You can first develop a framework that makes predictions and recommendations for SMEs to evaluate and help refine the models and recommend remediation actions. When you have built trust in the recommendations, then you can have the AI framework provide those recommendations directly to your z/TPF operators to be implemented under the supervision of your SMEs. When the framework has been refined to make highly reliable recommendations, you can take the final step of fully automating responses to various situations.

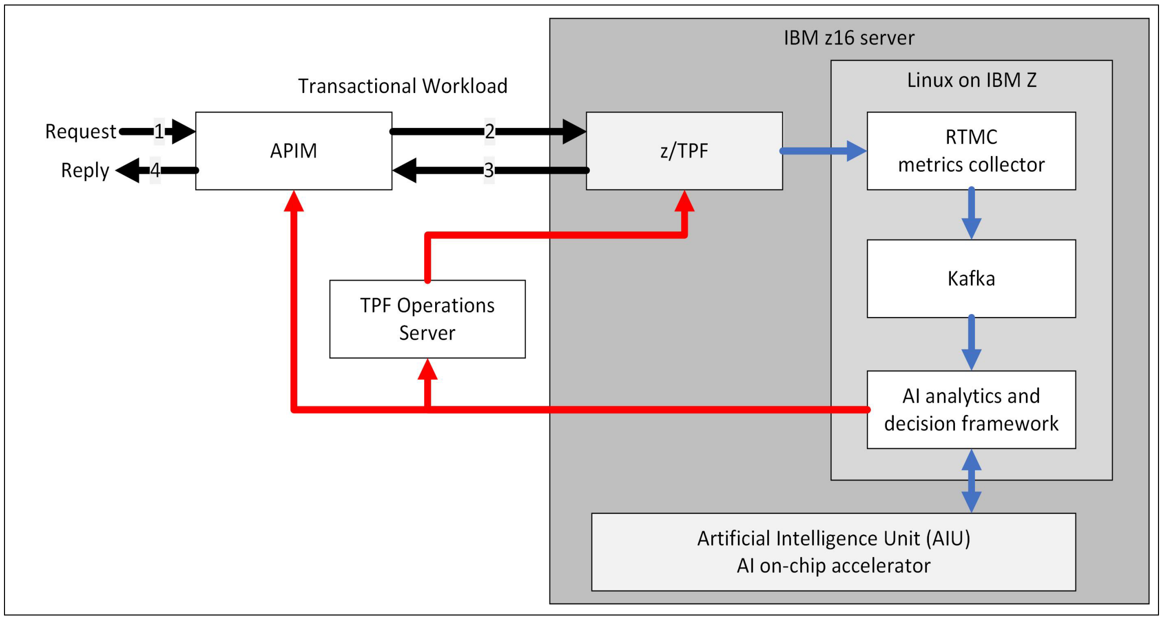


Figure 41 Leveraging AI to adjust your environment in real time

In addition to analytics that are based on Runtime Metrics Collection results, you can feed TPF Operations Server consoles to analytics packages in real time and leverage AI and ML to analyze, predict, and further automate situational reactions and proactive procedures to keep your z/TPF system running optimally.

VI. Making AI-driven decisions at scale

This entry point is focused on leveraging AI on the IBM Z platform to bring intelligence into applications and business transactions. In addition to using AI for anomaly detection in operations (as described in “z/TPF operations experience for leveraging AI insights” on page 40), AI also presents a huge opportunity to turn transaction data into actionable insights and actions that can amplify human capabilities, decrease risk, and increase the return-on-assets by achieving break-through innovations. For example, one use case that can be leverage is using AI in transaction processing for real-time fraud detection on IBM Z.

For your enterprise to be best positioned to leverage AI, you must access insights from historical data at scale, when and where it is needed, without impacting mission-critical workloads or SLAs. For most organizations, this situation is a challenge today. As indicated in *Operationalizing Fraud Prevention on IBM z16: Reducing Losses in Banking, Cards, and Payments*, only 10% of transactions in high-volume enterprise workloads go through real-time AI screening today. Still, AI continues to have a profound impact on the way enterprises do business, and adopting AI has become a business imperative.

AI on IBM Z

IBM strives to be an industry leader in enterprise AI solutions and enables clients to infuse AI in real time on IBM Z, which includes the following actions:

- Providing hardware and software capabilities to infuse AI in real time into applications.
- Enable deployment of AI models instead of developing and training IBM Z specific AI models, either through Open Neural Network Exchange (ONNX) or with popular open-source data science packages like TensorFlow or Spark.

For example, the IBM z16 processor has an industry-first, on-chip AI accelerator that is designed for high-speed and latency-optimized inferencing. It can accommodate 300 billion inference requests per day with a 1 ms response time. It provides consistent response times with optimized inference that can scale with IBM Z workloads and score every transaction while still meeting the most stringent application SLAs.

The AI ecosystem is optimized to leverage many open-source AI frameworks (like TensorFlow), which enable seamless integration of open-source products into the IBM Z platform. This infrastructure promotes the flexibility of building and training AI models in any framework on any platform, such as on-premises, public cloud, private cloud, or hybrid cloud, and then deploying the AI models onto the IBM Z platform.

Enhancing z/TPF applications to include AI inferencing

There is high-value potential for enterprises to enhance z/TPF applications to include AI inferencing. For example, consider the following value points:

- AI impacts many industries.

  In the financial sector alone, it has been estimated that $298 billion dollars are lost due to avoidable credit card transaction declines. Also, banks can score only 10% of transactions by using the existing fraud detection algorithms, which result in a significant revenue loss.

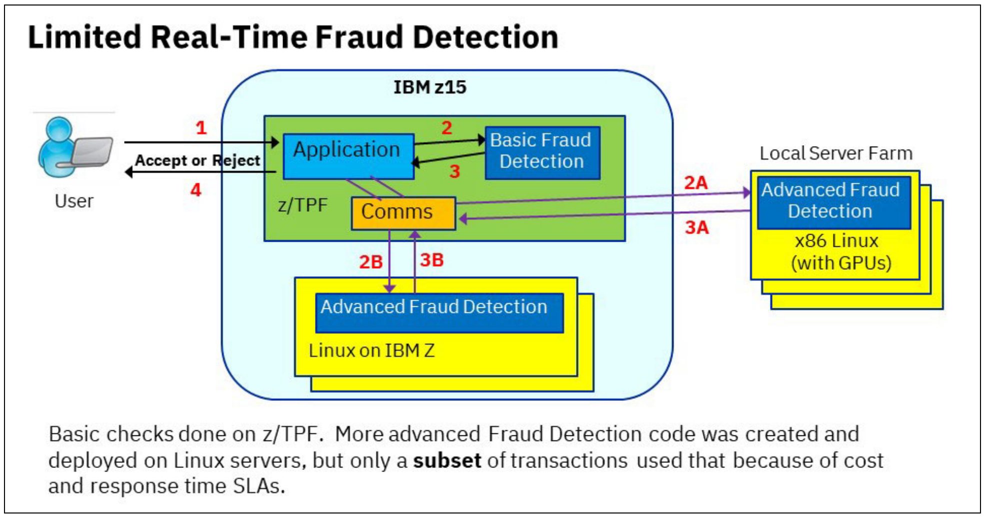
- On-premises fraud detection.

  As an example, an American bank was unable to score all transactions in real time to detect fraud patterns and prevent fraudulent transactions. When using an off-platform AI inference solution, clients could not score 80% of transactions due to SLA impacts. However, with IBM z16 on-chip AI accelerator, this American bank now can score 100% of transactions in real time and on-platform, and get response times within their SLAs, which result in significant cost savings and improved customer satisfaction.

- Clearing and settlement.

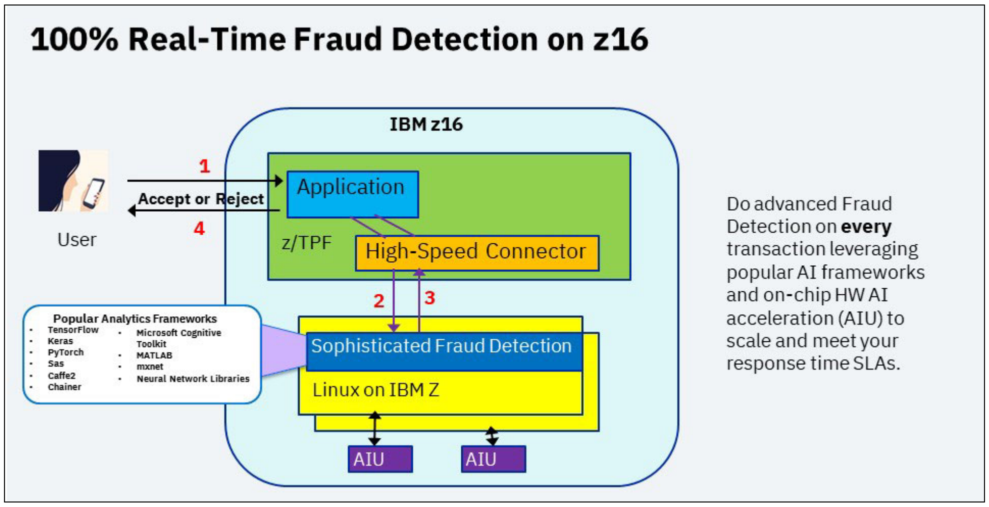
  Preventing fraudulent transactions is paramount to maintaining trust and improving the customer experience. A credit card processor needed to predict which trades and transactions have high risk exposures and foreign currency involvement before settlement, with no impact to SLAs and the batch process window. This company was able to augment its existing rules-based approach and embed an AI solution that uses TensorFlow for high performance on IBM Integrated Accelerator for AI and low latency scoring, avoiding costly consequences and regulatory violations.

z/TPF is aligned with the IBM AI strategy, and z/TPF can leverage the industry-first AI accelerator (AIU) on the z16 with AI frameworks running on Linux on IBM Z (see Figure 42). Using the example of fraud detection, an organization with this z/TPF environment can score 100% of transactions in real time by using advanced fraud detection algorithms (see Figure 43).



Basic checks done on z/TPF. More advanced Fraud Detection code was created and deployed on Linux servers, but only a **subset** of transactions used that because of cost and response time SLAs.

Figure 42 Limited real-time fraud detection



Do advanced Fraud Detection on **every** transaction leveraging popular AI frameworks and on-chip HW AI acceleration (AIU) to scale and meet your response time SLAs.

Figure 43 100% real-time fraud detection on an IBM z16

Similar to the clearing and settlement value point, this AI advancement in fraud detection can reduce risk. Before z16, a best practice was to use basic, rule-based fraud detection code running on z/TPF. With the growth of AI, more sophisticated analytics frameworks were developed that run on Linux that use deep learning neural network models, but because this workload was CPU-intensive, only a small subset of transactions could leverage this real-time scoring on Linux due to the cost. Now, with transactional AI inferencing, organizations can realize significant business benefits in ways that were not possible.

Reference architectures for modernization

In this section, we describe the following three z/TPF specific architecture models that can be used for modernizing your enterprise:

- Configuration architectures
- Using services in a hybrid cloud architecture
- Leveraging z/TPF data

Configuration architectures

Here, we examine configuration architectures and compare and contrast the characteristics of each one.

Single-image z/TPF systems

Some z/TPF applications were designed in a way where they could run on only one z/TPF system (LPAR) because they relied heavily on frequently updated memory tables or used other techniques that prevented the application from running in a z/TPF cluster. That one z/TPF system can be a single point of failure, but you can take steps to reduce the impact of potential outages.

A best practice is to have two IBM Z processors in the primary data center that are connected to the same DASD so that if your z/TPF system is running on a processor that is having issues, you can quickly move (perform an IPL) that logical z/TPF system onto the other local processor. The same DASD is used, so your database is current. The network traffic is still flowing into your primary data center, so only minor network updates are needed to direct the traffic to the z/TPF system running on the different physical processor in that data center.

Another best practice is to have at least one DR site so that if your primary data center becomes unusable for any reason, you can start z/TPF at the DR site. The DR site should be physically distant from your primary data center in case a natural disaster renders a geography unusable for days or weeks; your business can continue to operate. The DASD in the primary data center replicates updates to the DASD in the DR site. The DR site contains all the hardware (like an IBM Z processor) that is necessary to run z/TPF in that data center.

Figure 44 shows the environment during normal operations, where z/TPF is running on an IBM Z processor in the primary data center.

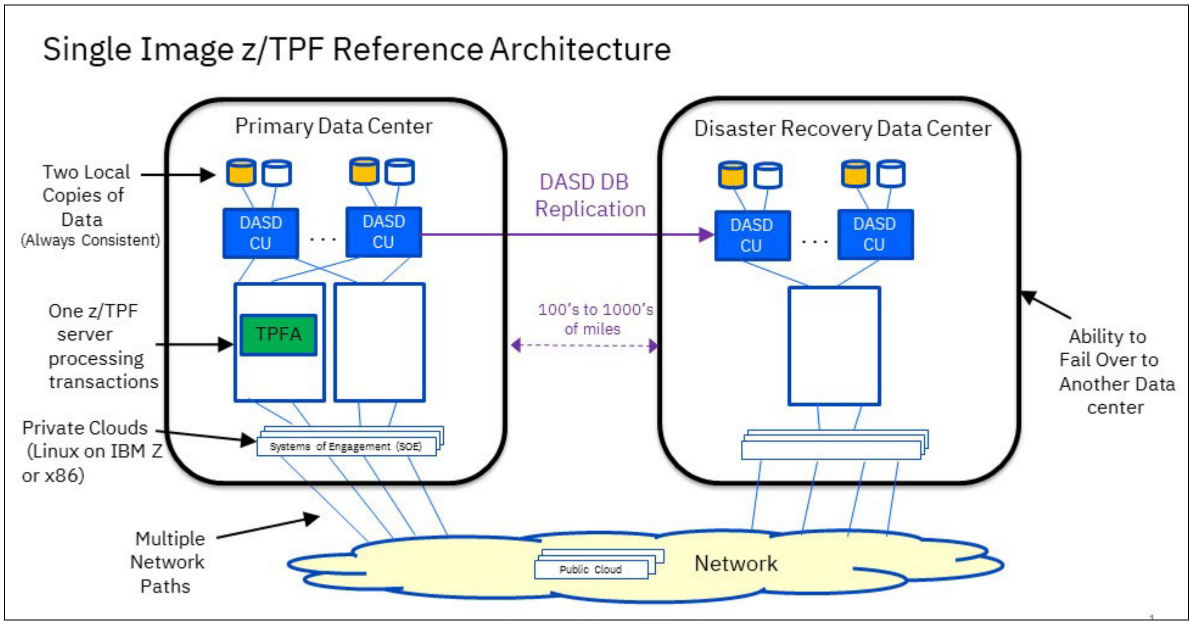

Single Image z/TPF Reference Architecture
Primary Data Center
Disaster Recovery Data Center
Two Local Copies of Data (Always Consistent)
DASD CU
DASD CU
DASD DB Replication
DASD CU
DASD CU
One z/TPF server processing transactions
TPFA
100's to 1000's of miles
Ability to Fail Over to Another Data center
Private Clouds (Linux on IBM Z or x86)
Systems of Engagement (SOE)
Multiple Network Paths
Public Cloud
Network

Figure 44 Single-image z/TPF reference architecture

Figure 45 shows a case where the main IBM Z processor fails and z/TPF is moved to run on the backup IBM Z processor in the primary data center.

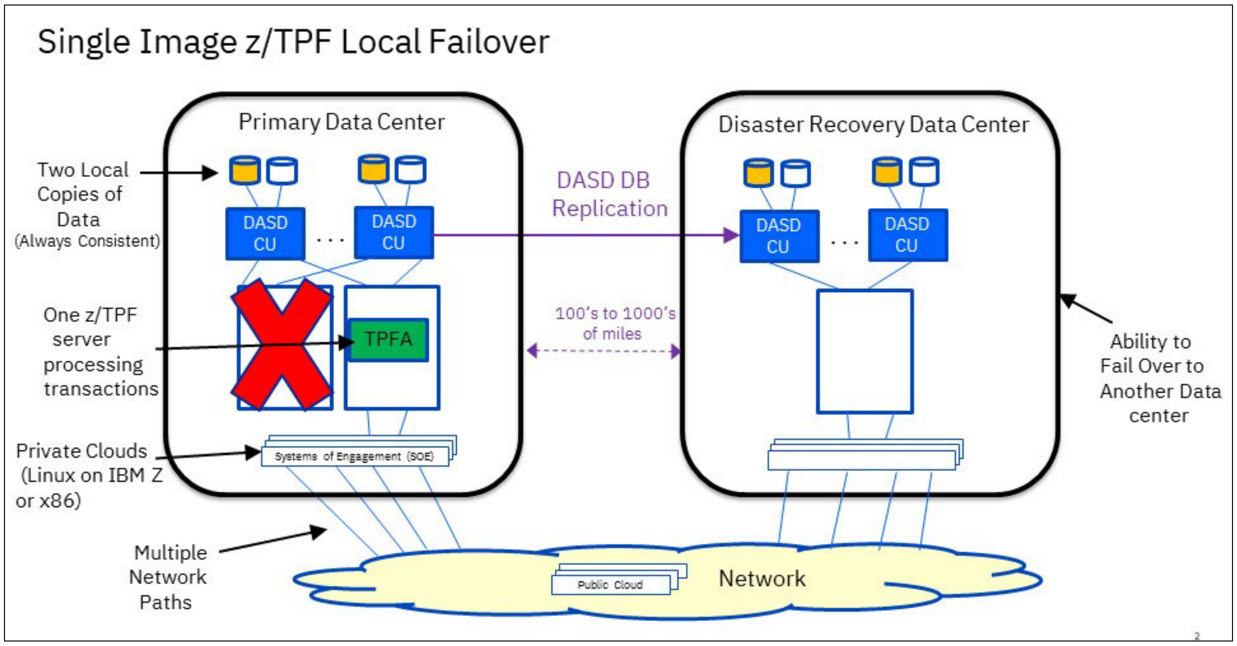

Single Image z/TPF Local Failover
Primary Data Center
Disaster Recovery Data Center
Two Local Copies of Data (Always Consistent)
DASD CU
DASD CU
DASD DB Replication
DASD CU
DASD CU
One z/TPF server processing transactions
TPFA
100's to 1000's of miles
Ability to Fail Over to Another Data center
Private Clouds (Linux on IBM Z or x86)
Systems of Engagement (SOE)
Multiple Network Paths
Public Cloud
Network

Figure 45 Single-image z/TPF local failure

Figure 46 shows a case where you lost the primary data center and z/TPF started in the DR data center to continue processing transactions.

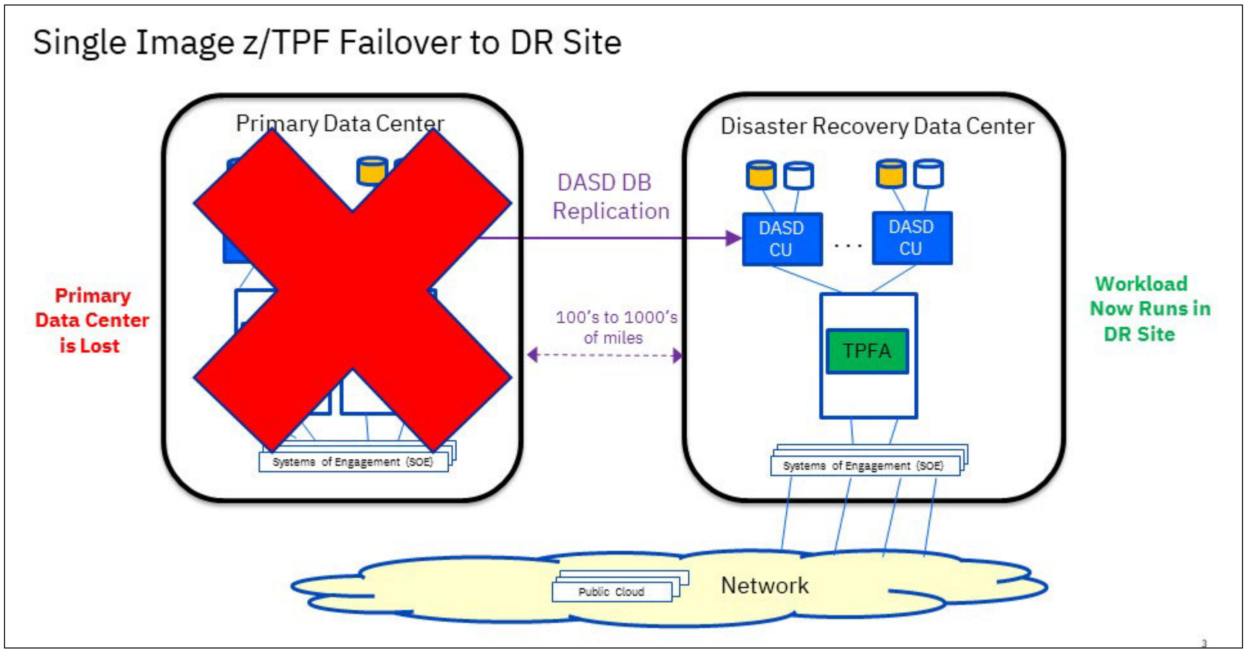
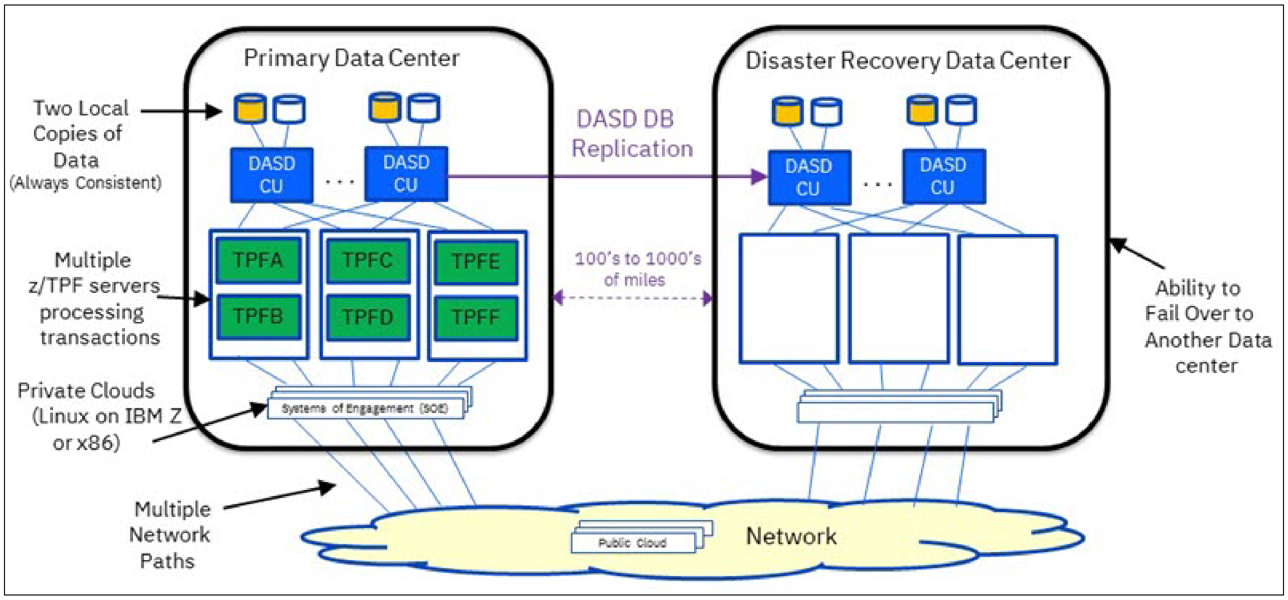

Single Image z/TPF Failover to DR Site
Primary Data Center
Systems of Engagement (SOE)
Primary Data Center is Lost
DASD DB Replication
100's to 1000's of miles
Disaster Recovery Data Center
DASD CU
DASD CU
TPFA
Systems of Engagement (SOE)
Workload Now Runs in DR Site
Public Cloud
Network

Figure 46 Single-image z/TPF failover to DR site

Active-active z/TPF cluster

z/TPF applications that were designed so that all shared data is on DASD can run in an active-active environment, which means multiple z/TPF LPARs simultaneously run on multiple IBM Z processors that all share the same DASD. This type of environment, which is referred to as a z/TPF loosely coupled complex, provides high availability within a single data center. You want to have enough CPU capacity so that if you lose one or two z/TPF instances or a single IBM Z processor, the remaining z/TPF instances can handle the entire workload. Even in an active-active environment, you still want to have one or more DR sites for the case where the primary data center becomes unusable.

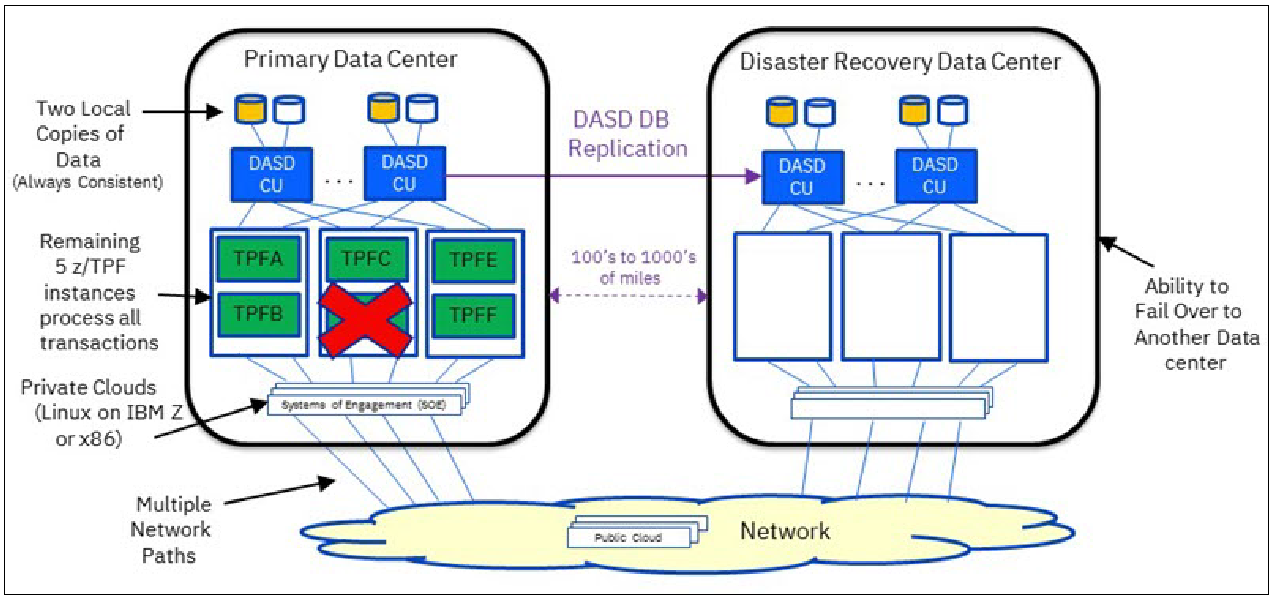
Figure 47 shows the environment during normal operations, where multiple z/TPF instances are running in the primary data center, all processing transactions and sharing the database.



Primary Data Center
Disaster Recovery Data Center
Two Local Copies of Data (Always Consistent)
DASD CU
DASD CU
DASD DB Replication
DASD CU
DASD CU
Multiple z/TPF servers processing transactions
TPFA
TPFB
TPFC
TPFD
TPFE
TPFF
100's to 1000's of miles
Ability to Fail Over to Another Data center
Private Clouds (Linux on IBM Z or x86)
Systems of Engagement (SOE)
Multiple Network Paths
Public Cloud
Network

Figure 47 z/TPF active-active reference architecture

Figure 48 shows the case where one z/TPF instance fails, but the remaining z/TPF instances have enough capacity to process all the transactions.



Primary Data Center
Disaster Recovery Data Center
Two Local Copies of Data (Always Consistent)
DASD CU
DASD CU
DASD DB Replication
DASD CU
DASD CU
Remaining 5 z/TPF instances process all transactions
TPFA
TPFB
TPFC
TPFE
TPFF
100's to 1000's of miles
Ability to Fail Over to Another Data center
Private Clouds (Linux on IBM Z or x86)
Systems of Engagement (SOE)
Multiple Network Paths
Public Cloud
Network

Figure 48 z/TPF active-active: One z/TPF instance fails

Figure 49 shows the case where you lost the primary data center and move the z/TPF cluster to run in the DR site.

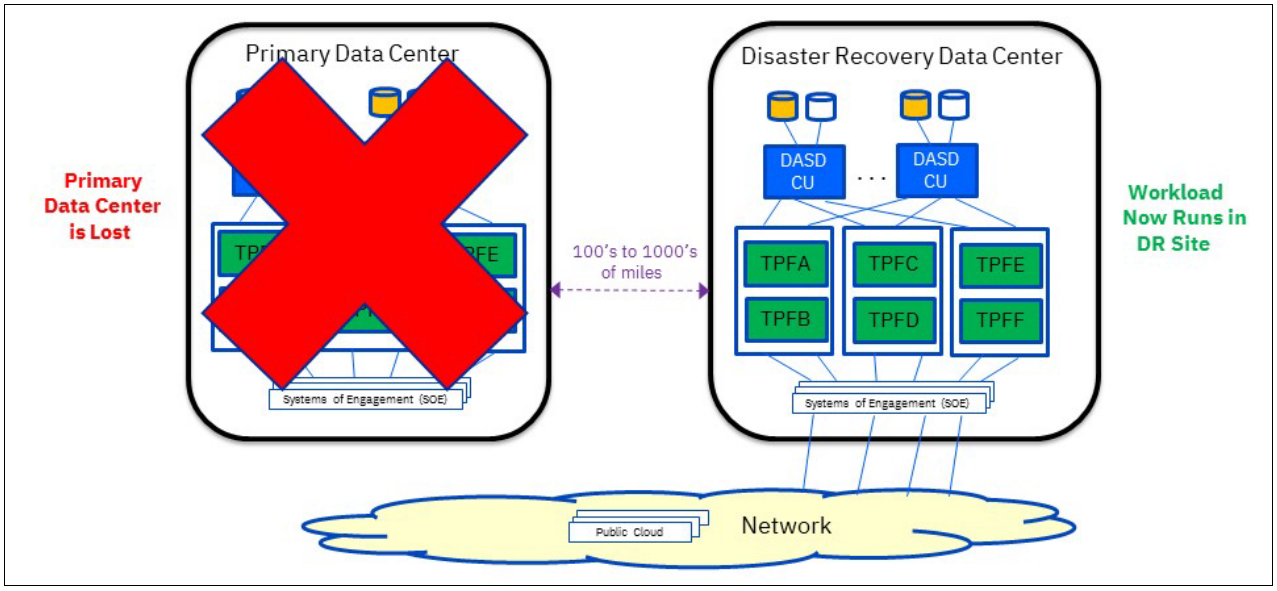

Primary Data Center is Lost
Primary Data Center
Systems of Engagement (SOE)
100's to 1000's of miles
Disaster Recovery Data Center
DASD CU
DASD CU
TPFA
TPFB
TPFC
TPFD
TPFE
TPFF
Systems of Engagement (SOE)
Workload Now Runs in DR Site
Public Cloud
Network

Figure 49 z/TPF active-active failover to DR site

Geographically dispersed active-active z/TPF systems

This architecture is where there are multiple z/TPF systems running in different data centers that can be thousands of miles apart. The z/TPF systems can be a single image or a loosely coupled configuration. Each z/TPF system has its own copy of the database on DASD and a transaction is processed locally on one z/TPF system. The database updates from that transaction are replicated asynchronously to the other z/TPF systems by user code. In this architecture, the separate databases are not always consistent, but are eventually consistent. In this database model, it is possible for applications to read old or stale data. In addition, the applications on different z/TPF systems can update the same database records concurrently, which require the user code to perform complex database conflict resolution. For any architecture that uses an eventually consistent database model, you must do a risk assessment to determine whether your business can tolerate reading old data at times. Although there are methods to reduce the likelihood of reading old data and data conflicts, this situation can still occur.

Figure 50 shows an example where there are three worldwide z/TPF systems processing transactions. User routing algorithms decide which z/TPF system processes a transaction and routes the transaction to that z/TPF system. The selection algorithm should route all related requests, such as all transactions for a given credit card number, to the same z/TPF system to reduce the likelihood of data conflicts. For example, a transaction is routed to and processed by the z/TPF system in the European Union (EU) data center. After the transaction is processed, a copy of all database updates that were made by that transaction are asynchronously sent to the other z/TPF systems in the United States (US) and Asia Pacific (AP). In this example, user code on the z/TPF systems in those other data centers update their copy of the database with the latest information.

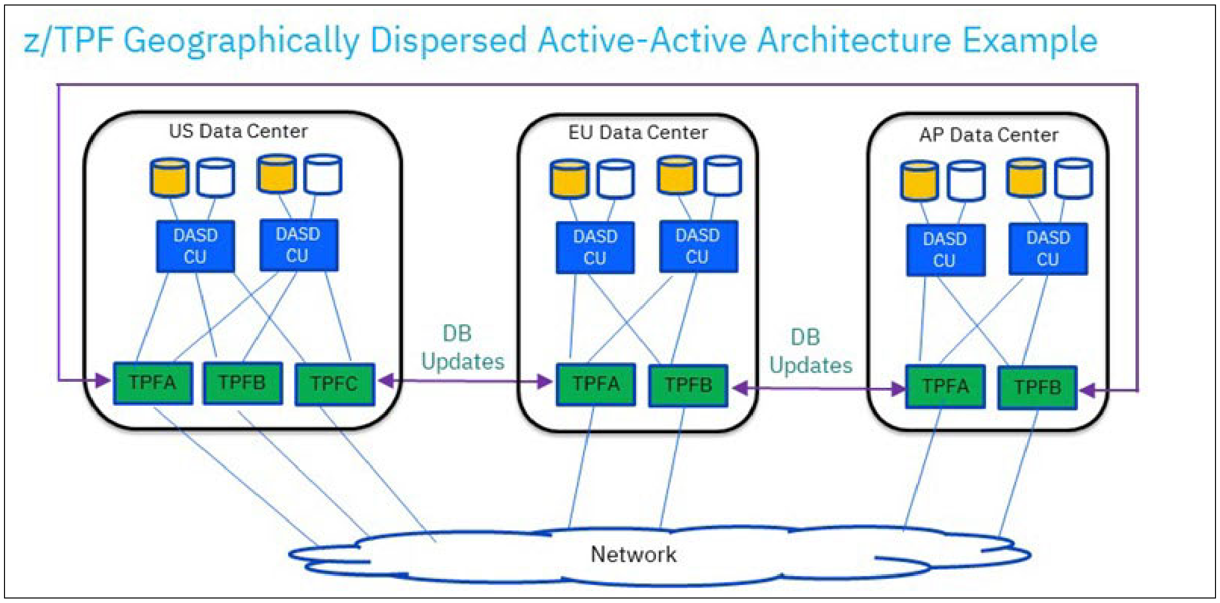


Figure 50   *z/TPF geographically dispersed active-active architecture example*

Using services in a hybrid cloud architecture

In this section, we describe how z/TPF provides and consumes services. Also, we describe the need for colocation of latency-sensitive aspects of your workload and the orchestration of multiple API calls.

Simplifying access from digital channels with APIs

The REST APIs that are exposed by the IBM Z platform can be integrated into hybrid cloud applications that are running on any cloud. You can REST-enable your existing z/TPF applications to extend your core services to be easily consumable by hybrid cloud applications that use the same standard protocols that are used on any other platform. Tools and schema-based data transformation leveraging the DFDL standard allows you to expose your existing z/TPF applications as services. An enterprise-wide APIM layer such as IBM API Connect® provides centralized access control and management of all services, including those services that are on z/TPF.

Figure 51 shows an example where services are on z/TPF and a pair of private clouds, with one service on Linux on Z and the other service on x86 Linux. Multiple public clouds and one private cloud consume those services. Access to all services goes through an API gateway, which provides centralized control. For services that are on z/TPF, the REST provider stack leverages DFDL to convert REST request data into the native application data format and convert the output native application data format into REST response data.

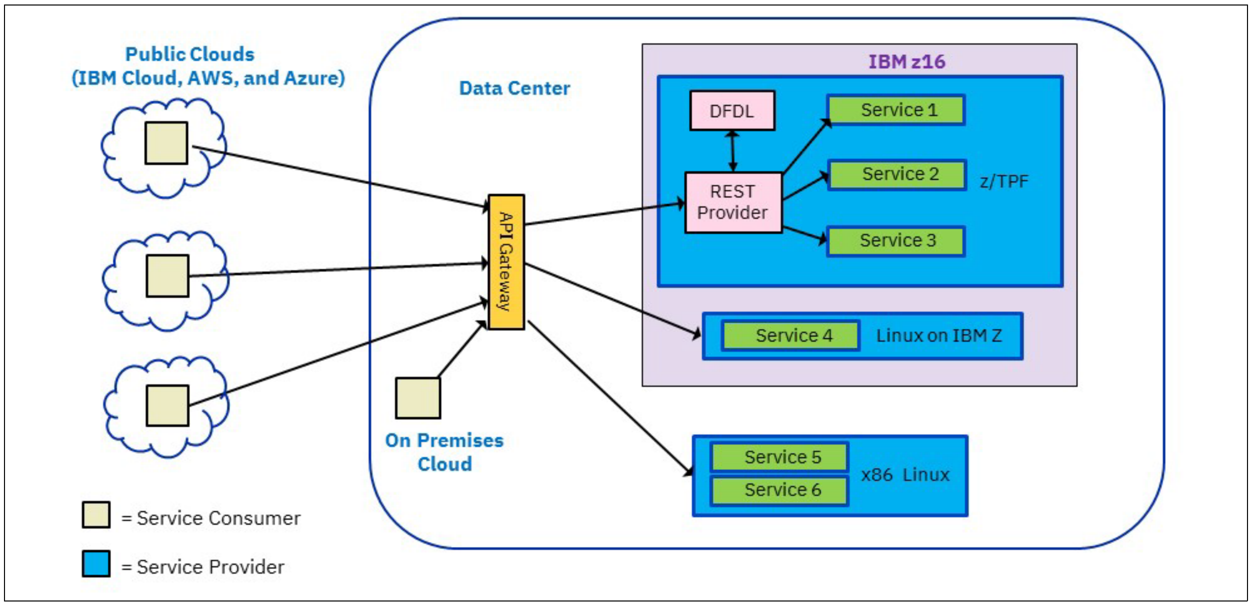


Figure 51 Reference architecture: Providing services

Simplifying access to digital channels with APIs

In addition to exposing z/TPF applications as services to hybrid cloud applications by using REST APIs, your z/TPF applications can consume external services to be one of the many service consumer platforms in your enterprise issuing REST APIs. REST allows you to make services available to z/TPF and other platforms by using a standard interface. An APIM layer provides centralized access control and management of all your services. Tools and schema-based data transformation leveraging the DFDL standard enable your existing z/TPF applications to consume external services.

Figure 52 shows an example where services are on multiple public clouds and on a pair of private clouds in your data center. There are many consumers of those services, but z/TPF is the only service consumer that is shown in this diagram. Access to all services goes through an API gateway, so you have centralized control of all services. When z/TPF applications issue REST APIs to consume services, the REST consumer stack leverages DFDL to convert the native application data format to the REST request messages and convert the REST response data that is received into the native application data format.

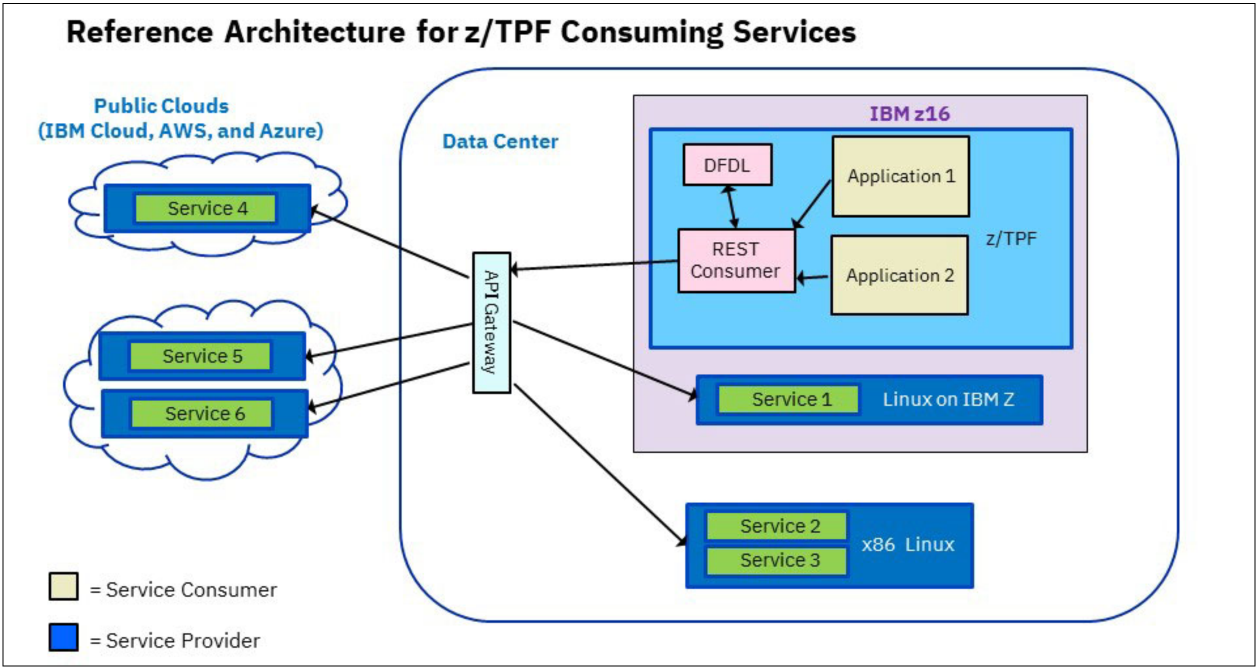


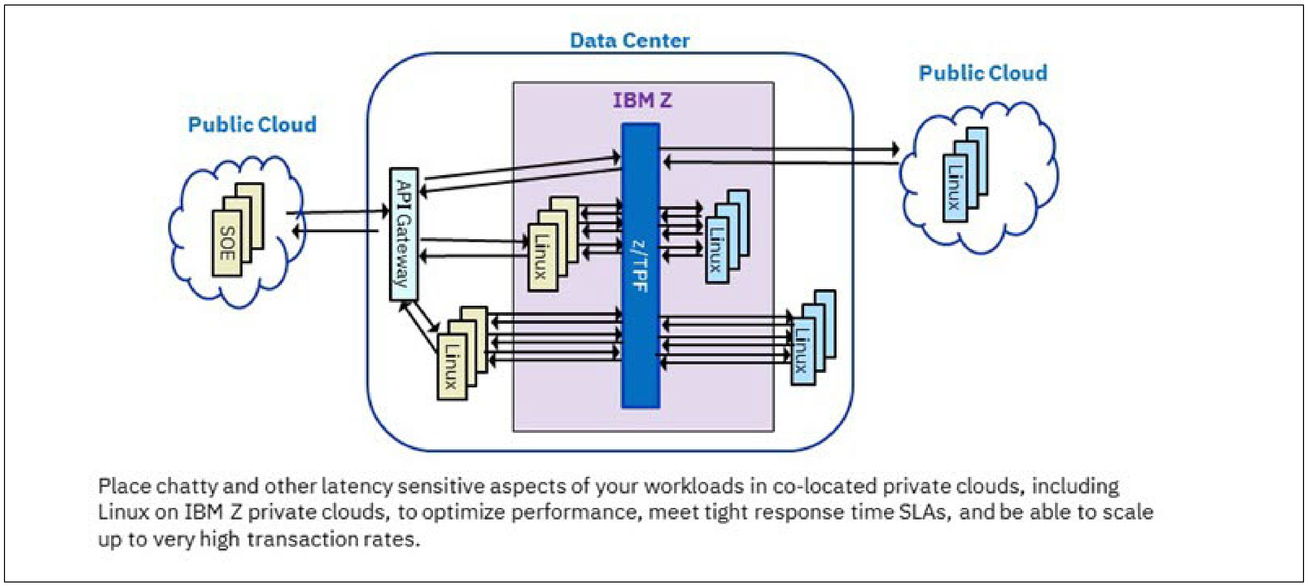
Figure 52 Reference architecture: z/TPF consuming services

Colocating latency-sensitive aspects of your workload

Conceptually, applications that consume and provide services by using REST APIs can be anywhere, but in reality, many real-time workloads with stringent response time SLAs require that certain components of the workload are physically close, such as colocated in the same data center. For example, to process a transaction, if client X and server Y are in different geographies, the latency of a single round-trip flow between components could be higher than your transaction response time SLA. A less obvious case is where the network latency between X and Y is lower than the response time SLA, but the workload involves multiple flows between X and Y such that the cumulative latency of all those flows exceeds the response time SLA. For these workloads, colocating components X and Y are a necessity, which does not necessarily mean that all aspects of a workload need to be colocated, but the latency sensitive aspects do.

In addition to meeting response time SLAs, the ability to scale up a workload to higher transaction volumes is another reason why certain components need to be colocated. For example, you might have a workload that obtains and holds a database lock for 20 ms across flows between components X and Y, but the network latency is over 18 ms. The application can process, at most, 50 transactions per second that use that same database record lock. If you colocate X and Y in the same data center, you reduce the network latency and the lock hold time to 2 ms; now, the same workload can now scale up to process 500 transactions per second that use the same database record lock.

Figure 53 shows an example of a hybrid cloud architecture where a workload is processing on z/TPF, public clouds, and private clouds (on Linux on IBM Z® and x86-based Linux). z/TPF provides and consumes services. For the latency sensitive aspects of the workload that are depicted by multiple flows between z/TPF and a cloud, those clouds are colocated within the same IBM Z processor or in the same data center as z/TPF.



Place chatty and other latency sensitive aspects of your workloads in co-located private clouds, including Linux on IBM Z private clouds, to optimize performance, meet tight response time SLAs, and be able to scale up to very high transaction rates.

Figure 53 Reference Architecture: Colocation for latency-sensitive aspects of your workload

For workloads that include latency-sensitive aspects, account for fail-over scenarios. If you move one aspect of a latency sensitive workload to another location, move all other latency sensitive aspects concurrently. Otherwise, components communicate over long distances and experience high network latency, which result in the problems that are described in this section. For example, Figure 54 shows a workload where the transaction starts with a public cloud calling z/TPF. Then, z/TPF makes multiple calls to on-premises private clouds (one on Linux on IBM Z, and the other on x86 Linux), which are latency sensitive flows. Then, z/TPF sends the response back to the public cloud. All the servers that process the latency-sensitive part of the workload are colocated in the same data center to achieve response time SLAs.

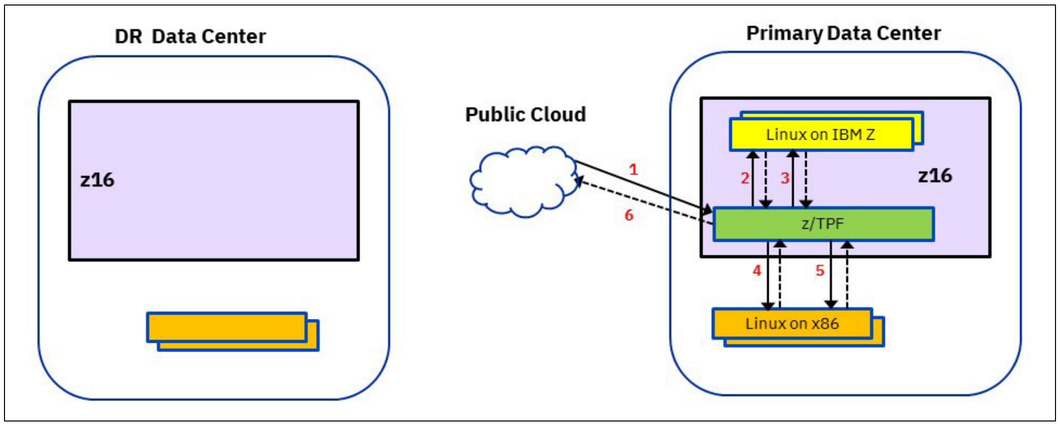


Figure 54 Workload running in primary data center

Suppose that the x86-based private cloud in the primary data center fails for whatever reason. Figure 55 shows what happens if you move only that part of the workload to the DR site. Now, flows between z/TPF and the x86-based cloud are across thousands of miles and incurring high latency. The result is that you are not able to meet your transaction response time SLAs.

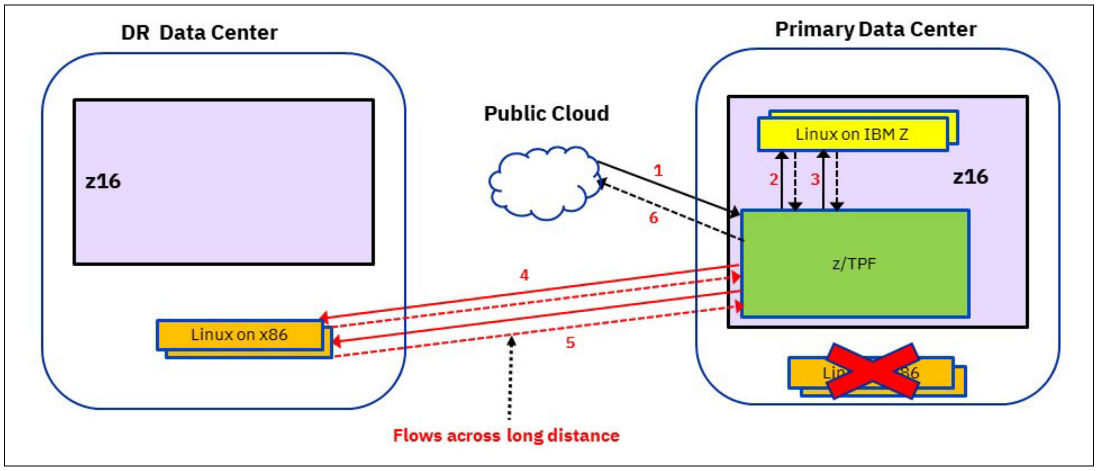


Figure 55 Failover of partial workload results in high latency

Figure 56 shows the correct action to take when the x86-based private cloud in the primary data center fails. All the latency-sensitive parts of the workload are moved to the DR site, which in this example means moving z/TPF and both private clouds to the DR site. The result is that you are once again able to meet your transaction response time SLAs.

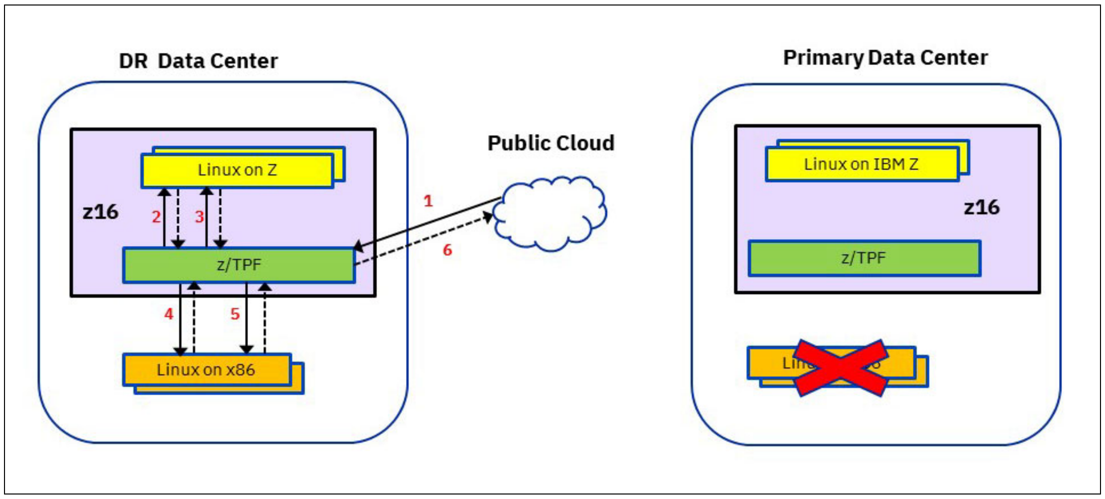


Figure 56 Proper failover: Moving the entire latency-sensitive part of the workload

Orchestrating multiple API calls

When modernizing a monolithic application, it is a best practice to break the application into small pieces that are called microservices. Smaller components with well-defined interfaces make it easier to maintain the code and to reuse components.

If you had a remote client that used a monolithic application through a single flow between client and server and now the application on the server is broken up into 20 microservices, there are performance issues if the remote client now makes 20 REST API calls to the server. These issues include CPU costs on both the client and server platforms and transaction latency issues with 20 round trips between the client and server. A common method to get the benefits of microservices but still achieve acceptable runtime performance is to create an orchestration layer that exposes a single macroservice to the client and then drives the necessary microservices to the servers where the microservices are. If you write that orchestration layer in Java, you can deploy it on z/TPF to maximize performance so that there is a single call from the remote client to z/TPF, and then multiple, optimized local calls between the orchestration layer and the different microservices on z/TPF.

Figure 57 shows an orchestration layer that is written in Java that exposes one macroservice to the remote service consumer that calls multiple microservices on z/TPF. If you deploy that orchestration on a colocated private cloud that is near the z/TPF system, you might be able to achieve acceptable transaction response times for your SLAs but still incur the CPU overhead of multiple REST flows on z/TPF and Linux. A better implementation is to deploy that orchestration layer on z/TPF so that there is a single, external flow, and all the microservice calls are optimized internal local calls.

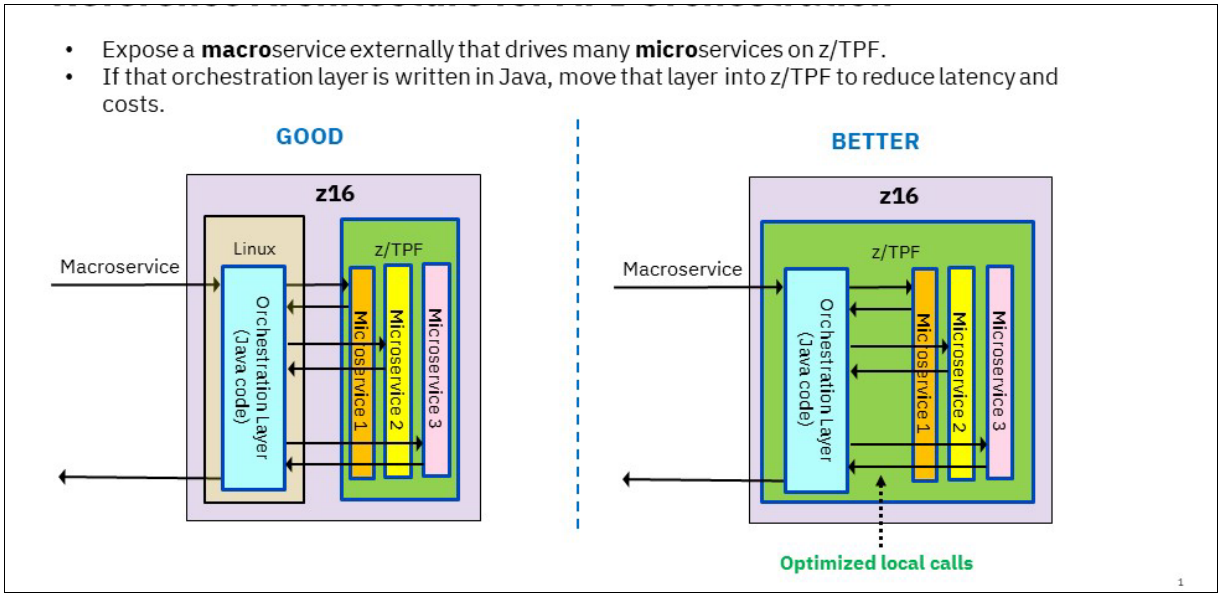

- Expose a **macro**service externally that drives many **micro**services on z/TPF.
- If that orchestration layer is written in Java, move that layer into z/TPF to reduce latency and costs.

Figure 57 Reference architecture for API orchestration

Leveraging z/TPF data

In this section, we describe standard remote access to z/TPF data, and the benefits of leveraging event-driven architectures.

Standard remote access to z/TPF data

Usually, client applications call services on z/TPF that access and update multiple databases. The service returns the relevant information to the client in a response, such as through a REST API. The client does not need to understand what information is in which physical databases. Suppose that a remote client needs the latest copy of a specific database record on z/TPF. If the data is in a z/TPF Database Facility (z/TPFDF) database, you can use the MongoDB interface for z/TPF. The MongoDB interface for z/TPF enables the remote client to read the latest copy of the record by using a standard MongoDB client. The client application thinks that it is communicating with a MongoDB server and has no idea that it is connected to a z/TPF system. The z/TPF system code does all the interfacing to z/TPFDF and the data format transformation for the MongoDB response. User code on z/TPF is not needed. This mechanism enables cloud-based applications to access and optionally update the same z/TPFDF databases that local applications on z/TPF are using.

Figure 58 shows applications in public and private clouds that use a standard MongoDB client to access data on z/TPF. The MongoDB interface layer on z/TPF leverages DFDL to convert the data from z/TPFDF data format to MongoDB data format. The databases that are accessed by remote MongoDB clients are the same databases that local applications on z/TPF access, so you can share data for multiple purposes.

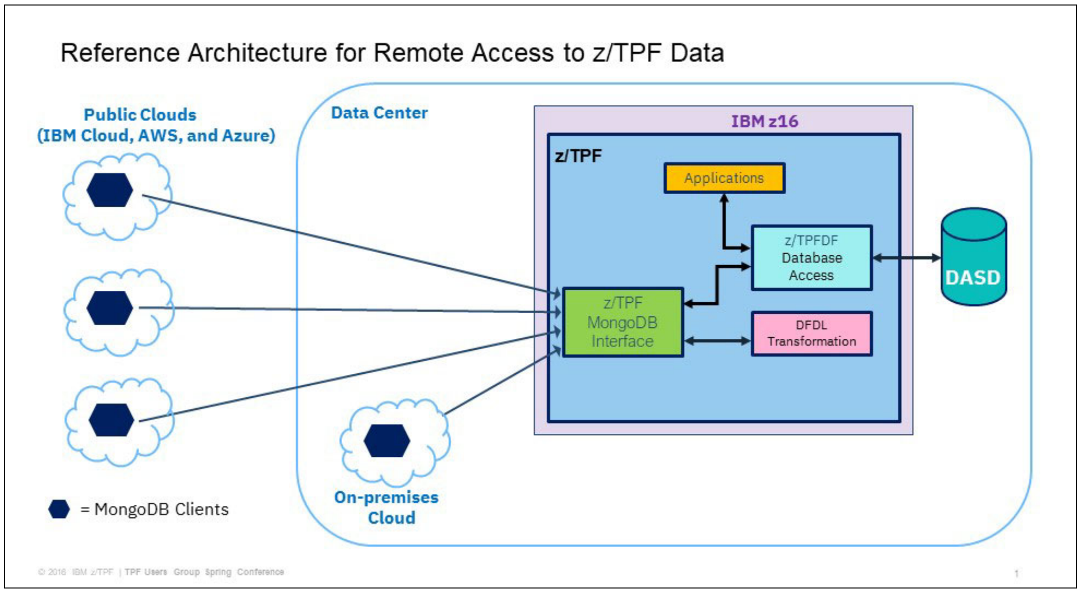


Figure 58 Remote access to z/TPF data by using a standard MongoDB client

z/TPF contains system of record (SOR) data that is often needed by other parts of your enterprise. Use z/TPF data events so that when your applications update certain databases, a copy of the data is sent to the remote data consumers that need that data. For security and compliance reasons, use the filtering capabilities of data events and DFDL so that each remote data consumer gets only the parts of the data for which they have a business need. Use the data transformation capabilities of DFDL to convert the data to a format that is more easily consumable externally, such as XML or JSON. You can choose a transport mechanism for each remote destination that has the appropriate qualities of service for that data, like guaranteed delivery and high volume.

Figure 59 on page 60 shows z/TPF applications updating databases on z/TPF as part of processing transactions. Based on the database definitions, whenever a specific database is updated, a copy of that data is placed on a local queue, and the application continues processing the transaction. Asynchronously, the system processes the local queue, and for each remote data event consumer that is defined for this event, the system calls a customizable dispatch adapter. The dispatch adapter filters and formats the data, and then uses the transport that the user defined to send that data to the remote platform.

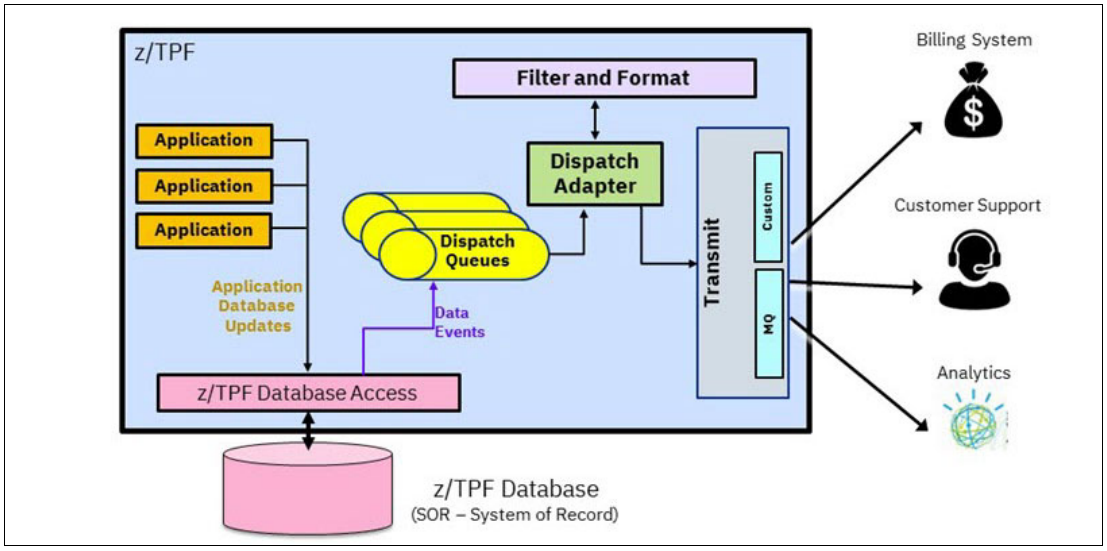


Figure 59 Architecture of z/TPF data events

Leveraging an event-driven architecture

Many times when a z/TPF application is processing a transaction, a condition is detected that requires sending a notification (an event) to one or more other platforms. There are two recommended methods for doing this task on z/TPF:

- Use z/TPF signal events to format and send the event data to one or more remote event consumers.
- Use Kafka to post the event in a Kafka stream.

Figure 60 shows z/TPF applications issuing signal event APIs to generate events whose data is placed on a local queue, and the application continues processing the transaction. Asynchronously, the system processes the local queue, and for each remote data event consumer that is defined for this event, the system calls a customizable dispatch adapter. The dispatch adapter filters and formats the data, and then uses the transport that the user defined to send that data to the remote platform.

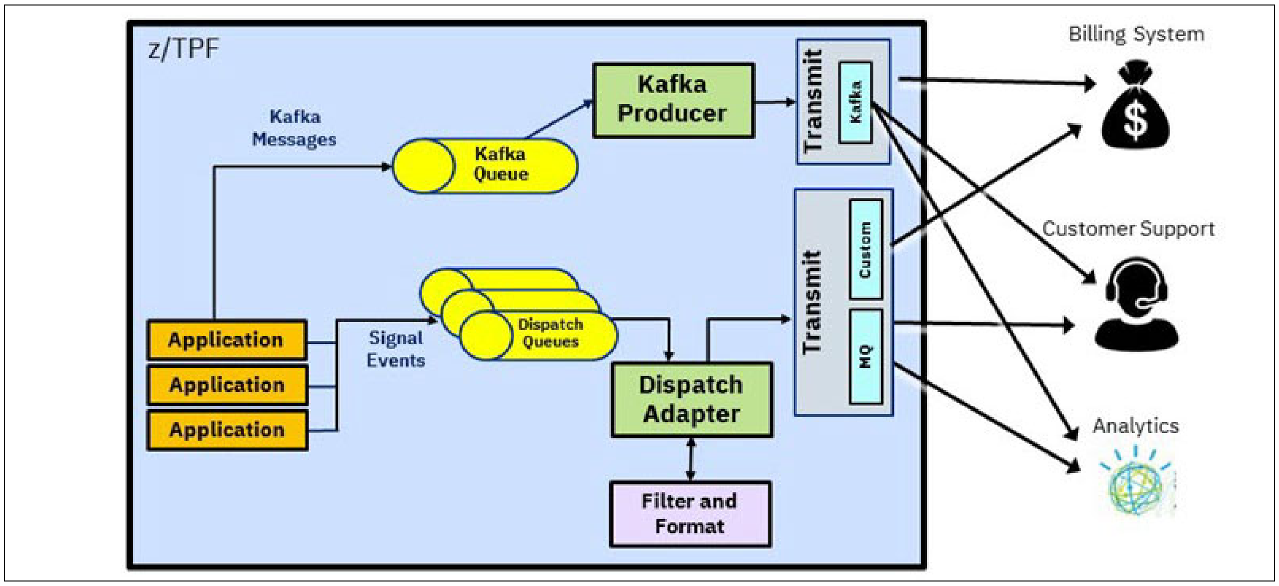


Figure 60 z/TPF applications generating events

Figure 60 on page 60 also shows z/TPF applications adding messages to a local IBM MQ queue that contain data that you publish to Kafka streams. The Kafka producer layer on z/TPF monitors and processes that local queue, and builds and sends the messages to the target Kafka cluster.

z/TPF Modernization Patterns

There are dozens of ways that you can modernize your z/TPF environment. This section outlines the different types of modernization patterns and the capabilities that you can leverage to achieve those goals. Choose the patterns that best fit your needs. Table 1 to Table 5 on page 62 provide the current z/TPF modernization patterns.

Table 1 *z/TPF Application Modernization Patterns*

Pattern	Technologies that can be leveraged for this pattern
Create a hybrid cloud architecture through fit for purpose.	Leverage strengths of both IBM Z and cloud.
Expose existing core services as APIs.	REST provider, OpenAPI, and DFDL
Extend apps to consume services through APIs.	REST consumer, OpenAPI, and DFDL
Refactor monolithic application into discrete services.	REST and OpenAPI
Progressively modernize old (assembler language) applications.	C/C++ and Java
Identify and eliminate dead code.	Active Program Detection Tool
Replace custom transports with standard middleware.	REST, HTTP, IBM MQ, and Kafka
Leverage open source rather than write from scratch.	Open-source Java Packages
Create repeatable automated regression test suites.	z/TPF Automated Test Framework
Identify gaps in test coverage.	Code Coverage Tool
Train new developers on how apps are being used.	z/TPF Message Analysis Tool

Table 2 *z/TPF Data Modernization Patterns*

Pattern	Technologies that can be leveraged for this pattern
Access z/TPF data remotely by using a standard DB client.	MongoDB interface for z/TPF
Feed consumable z/TPF data in real time to data consumers.	Business Events, DFDL, XML, JSON, and Kafka
Convert old style DBs to a better and more powerful DB access method.	z/TPFDF
Replace custom data transformation with a schema-driven process.	DFDL
Augment data with transactional context to gain additional business analytics insights.	Business events and AI
Create smarter applications leveraging more data that is available at zero latency (cached in memory).	z/TPF DB Cache (VFA), z/TPF Logical Record Cache, and terabytes of memory that is available on IBM Z

Table 3   z/TPF Security Modernization Patterns

Pattern	Technologies that can be leveraged for this pattern
Leverage pervasive encryption for all data in flight and at rest.	z/TPFDF Automatic DB Encryption, TLS, and IBM Z hardware crypto acceleration
Control access through a centralized APIM.	Enterprise APIM
Create a crypto inventory for audit and compliance.	z/TPF Compliance Capability
Prepare for quantum attacks NOW!	Use quantum-safe algorithms to protect data.
Back up your data in a data vault.	IBM DASD Safeguarded Copy
Score 100% of financial transactions in real time.	AI framework on Linux on IBM Z that uses z16 AIU
Filter data that is sent out by destination based on need to know.	DFDL
Monitor and validate operator activity for insider threat exposure.	TPF Operations Server, AI, and ML

Table 4   z/TPF Operations Modernization Patterns

Pattern	Technologies that can be leveraged for this pattern
Automate as much as possible.	TPF Operations Server
Monitor system and application health in real time.	Runtime Metrics Collection and AI
Instrument applications for faster problem determination.	Runtime Metrics Collection and AI
Instrument applications for more accurate capacity planning.	Runtime Metrics Collection
Instrument applications to create baseline data for predictive analytics.	Runtime Metrics Collection, AI, and ML
Add CPU capacity on demand for unexpected workload spikes.	Dynamic CPU support
Use additional CPU capacity to handle a workload surge following an unplanned outage.	System Recovery Boost (SRB)
Run utilities anytime without impacting transactions.	z/TPF Low-Priority Utility Support

Table 5   z/TPF Performance Modernization Patterns

Pattern	Technologies that can be leveraged for this pattern
Colocate latency-sensitive applications.	Linux on IBM Z LPARs next to z/TPF LPARs, and private (on-premises) clouds
Expose a macroservice externally, and then use orchestration to drive microservices internally.	REST, Java, and z/TPF Optimized Java to C and C to Java interface
Cache entire databases in memory.	z/TPF DB cache (VFA), and terabytes of memory that is available on IBM Z
Cache resource-intensive calculation results in memory.	z/TPF Logical Record Cache, and terabytes of memory that is available on IBM Z

Pattern	Technologies that can be leveraged for this pattern
Compress large messages.	z/TPF IBM MQ, z/TPF HTTP Server, zlib, and IBM Z on-chip hardware compression accelerator
Identify inefficiencies in your applications.	z/TPF Software Profiler z/TPF Message Analysis Tool
Deep learning at scale.	AI framework on colocated Linux on IBM Z that uses z16 AIU
Use DR hardware to do periodic, near full-scale testing.	zBuRST

The choice of process execution model matters

There are fundamental architectural differences between z/TPF and a cloud environment that enable z/TPF to scale up to handle hundreds of thousands of transactions per second while still maintaining SLAs. This section explains in detail one of those key differences, that is, the z/TPF process execution model.

Single process per transaction approach to transaction processing

On most platforms, there are separate, long-running, and multi-threaded processes that each handle different functions. In z/TPF, each typical transaction runs in its own single-threaded process, and all application and system components that are involved with processing that transaction share application memory. The most significant difference between these two models, as shown in Figure 61, is efficiency in terms of latency and CPU consumption.

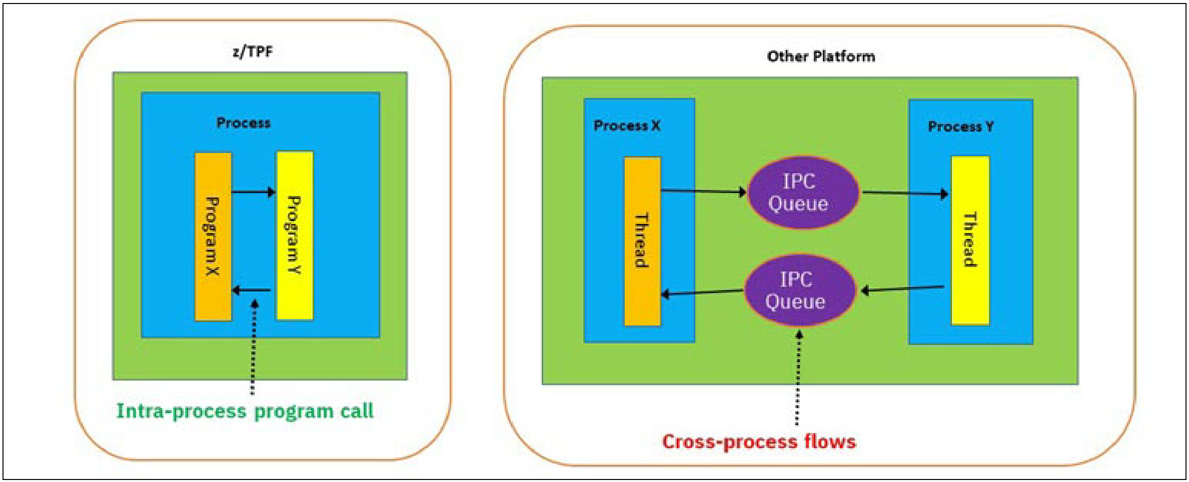


Figure 61 Local application call comparison

If program X must call program Y on z/TPF, the following steps occur:

1. Program X calls program Y directly in the same process. None of the data that program Y needs must be copied because programs X and Y both have access to the same data in memory.
2. Program Y gains control instantly and performs the requested task.

3. Program Y returns to program X. None of the response data that is needed by program X needs to be copied because program Y built that data in memory that is shared by programs X and Y.
4. Program X continues processing. There is no loss of control when one program calls another program or returns to the caller program.

If program X must call program Y on other platforms, then the following steps occur:

1. The thread in process X adds an item to an inter-process communication (IPC) queue. All the data in process X that process Y needs to perform the requested task is copied into the IPC item.
2. A thread in process Y is scheduled.
3. The thread in process X is suspended.
4. The thread in process Y is dispatched and reads the entry from the IPC queue, which copies the data that is passed from process X into process Y's memory.
5. The thread in process Y performs the requested task.
6. The thread in process Y adds an item to an inter-process communication (IPC) queue. All the response data that process X needs is copied into the IPC item.
7. The thread in process X is scheduled.
8. The thread in process Y is suspended.
9. The thread in process X is dispatched and reads the entry from the IPC queue, which copies the data that is passed back from process Y into process X's memory.
10. The thread in process X continues processing.

As the complexity of transactions grows, the benefits of the z/TPF model of a single process per transaction increases significantly. For example, Figure 1 on page 3 shows a transaction that is handled by another platform by a single server with separate processes for the TCP/IP stack, middleware, applications, and database. With other platforms, every time a transaction flows across the different processes, separate threads must make calls over IPC queues. As shown in Figure 2 on page 4, the z/TPF architecture is much more efficient in terms of CPU consumed (no IPC queues, copying of data, or process switching is needed), and latency (no waiting for separate threads to get dispatched).

There are also security and resiliency advantages when each transaction runs in its own process (the z/TPF model) versus an architecture that has several long-running processes to process multiple transactions concurrently.

In the z/TPF model, there is increased security because each transaction is isolated in its own process and can see only its own data, not the data of any other transaction. In the multi-threaded, long-running process model on other platforms, all threads within a process share the memory, so one transaction can see the data of other transactions that are handled by other threads in the same process.

There also is an increase in system resiliency with the z/TPF model, as shown in Figure 62 on page 65. In a long-running process model on other platforms with multiple threads handling concurrent transactions, an application logic error processing a transaction in one thread can cause the entire process to fail. As a result, all transactions running in that process are impacted, and subsequent transactions might be delayed while the long-running process is restarted. In this example, transactions that fail are not only the ones that were running in Process 2. Transactions that were running in Processes 3, 4, 5, and 6 also eventually fail because when they try to return to Process 2, that process no longer exists. In the z/TPF model, because each transaction is isolated with its own process, if there is an application logic error while processing that transaction, only that transaction is impacted (all other transactions continue processing).

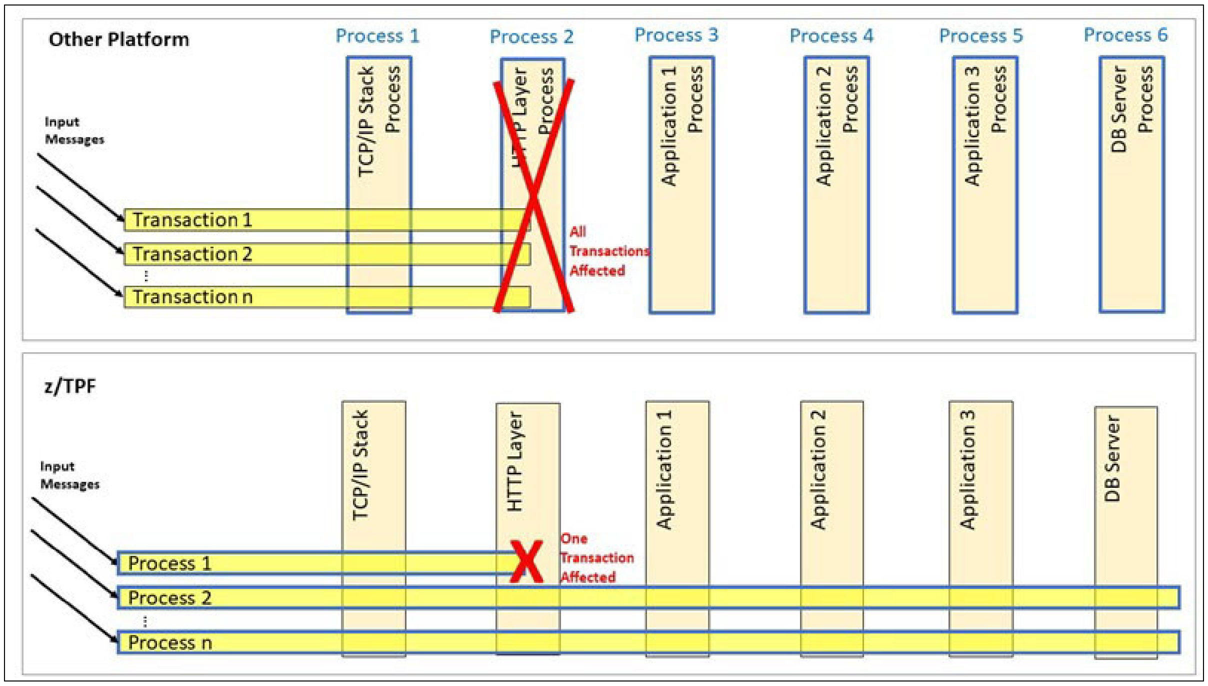


Figure 62 Resiliency of the z/TPF model

Another resiliency advantage of the z/TPF model is its ability to better handle workload spikes. On other platforms with long-running processes, each process has a certain number of threads that dictate how much workload per second that process can handle. When there are spikes in workload volume, more instances of the application might need to be started in new processes. If your workload monitoring tools do not detect and react fast enough to that situation or if starting those new processes takes several seconds, the situation can impact the ability to process all transactions in a timely manner and meet SLAs.

The z/TPF model does not use pre-existing, long-running application processes, so you do not need to be concerned about the following items:

- How many processes are running
- How to tune the number of threads in each process
- Starting new, long-running processes when needed
- Tearing down processes when they are no longer needed

Instead, each transaction runs in a new, single-threaded process that is created instantly; processes that one transaction; and exits when complete. For example, when a server polls the network, usually it receives up to 20 new messages, but then the next time it polls, it receives 250 new messages. On other platforms, if there are not 250 application threads already created and available to handle new work, some of those new 250 messages might not be processed in a timely manner. A z/TPF system immediately creates 250 new single-threaded application processes so that there is no impact when processing surges of traffic.

Yet another advantage of the z/TPF model is minimizing the CPU footprint that is needed to process your workloads. In a distributed architecture, each application typically runs in its own long-running, multi-threaded process. Each process can handle a certain number of transactions per second. An application that must handle large volumes of transactions requires multiple processes that are spread across many server nodes. Typically, those server nodes are under-utilized, running at 20 - 50% CPU busy. For example, in the upper part of Figure 63, you have applications, potentially in different clouds, where there are 100 servers for Application 1, 200 servers for Application 2, and 300 servers for Application 3.

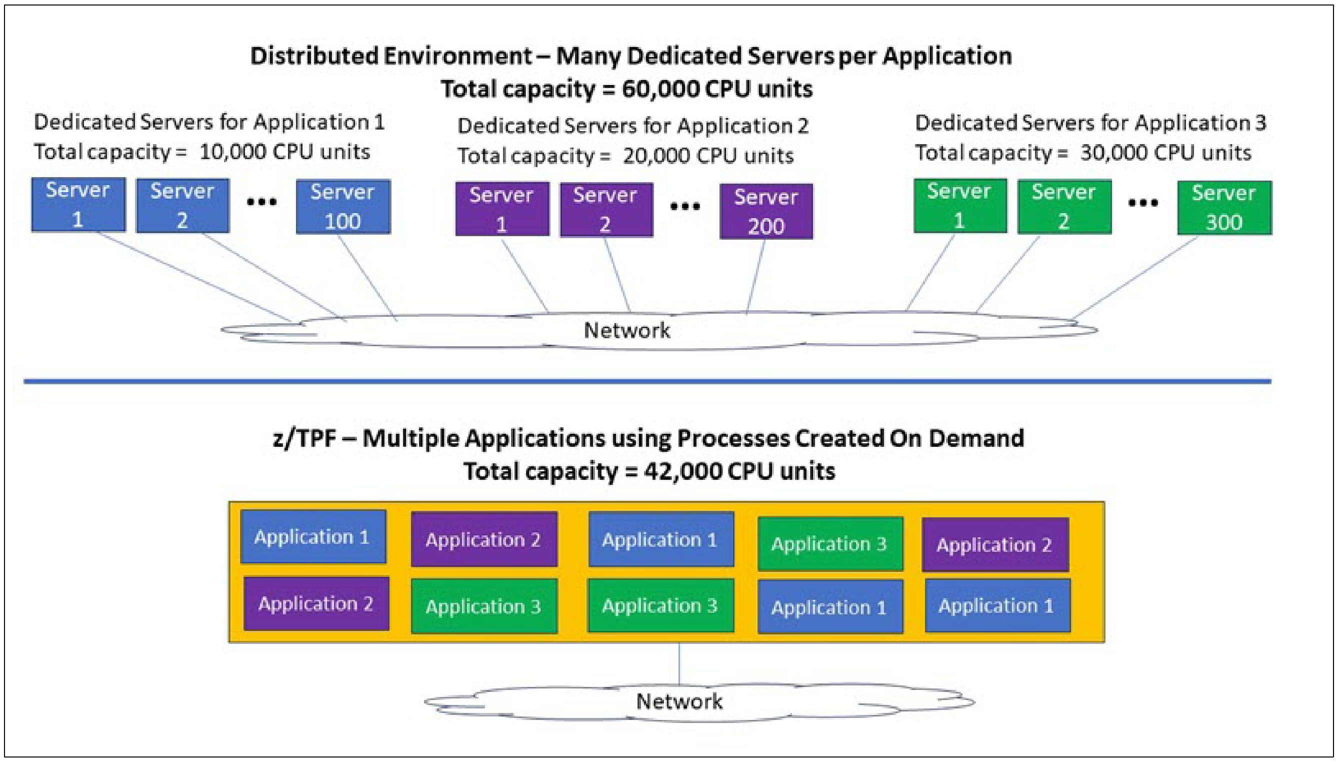
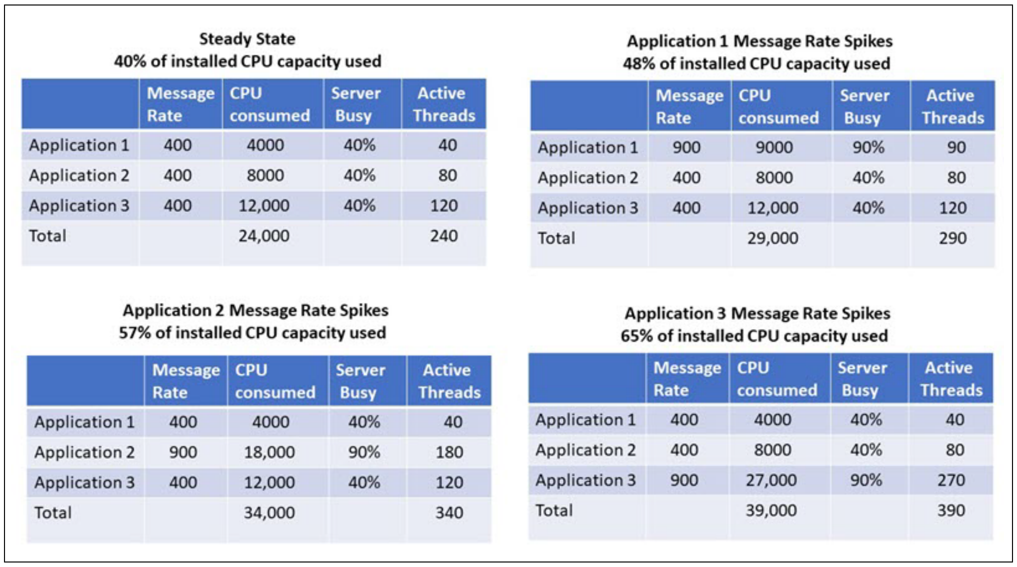

Distributed Environment – Many Dedicated Servers per Application
Total capacity = 60,000 CPU units
Dedicated Servers for Application 1
Total capacity = 10,000 CPU units
Server 1
Server 2
Server 100
Dedicated Servers for Application 2
Total capacity = 20,000 CPU units
Server 1
Server 2
Server 200
Dedicated Servers for Application 3
Total capacity = 30,000 CPU units
Server 1
Server 2
Server 300
Network
z/TPF – Multiple Applications using Processes Created On Demand
Total capacity = 42,000 CPU units
Application 1
Application 2
Application 1
Application 3
Application 2
Application 2
Application 3
Application 3
Application 1
Application 1
Network

Figure 63 Less CPU is required on z/TPF to handle multiple workloads

There is enough capacity where each application can process up to 1000 messages per second. Processing one message by application 1 consumes 10 CPU units, processing one message by application 2 consumes 20 CPU units, and processing one message by application 3 consumes 30 CPU units. The workload for each application spikes at different times of the day, so when one application is processing 900 messages per second, the other two applications are each processing only 400 messages per second.

Figure 64 shows the CPU that is consumed and active threads for the different message mixes. In steady state, only 40% of the installed CPU capacity (60,000 CPU units) is used, and at most 65% of the installed CPU capacity is ever used.



Steady State
40% of installed CPU capacity used

	Message Rate	CPU consumed	Server Busy	Active Threads
Application 1	400	4000	40%	40
Application 2	400	8000	40%	80
Application 3	400	12,000	40%	120
Total		24,000		240

Application 1 Message Rate Spikes
48% of installed CPU capacity used

	Message Rate	CPU consumed	Server Busy	Active Threads
Application 1	900	9000	90%	90
Application 2	400	8000	40%	80
Application 3	400	12,000	40%	120
Total		29,000		290

Application 2 Message Rate Spikes
57% of installed CPU capacity used

	Message Rate	CPU consumed	Server Busy	Active Threads
Application 1	400	4000	40%	40
Application 2	900	18,000	90%	180
Application 3	400	12,000	40%	120
Total		34,000		340

Application 3 Message Rate Spikes
65% of installed CPU capacity used

	Message Rate	CPU consumed	Server Busy	Active Threads
Application 1	400	4000	40%	40
Application 2	400	8000	40%	80
Application 3	900	27,000	90%	270
Total		39,000		390

Figure 64 CPU consumption in a distributed environment

The lower half of Figure 63 on page 66 shows the three applications (workloads) running on a single z/TPF system. Figure 65 shows the different message mixes for these applications on z/TPF. In the z/TPF model, there are no pre-defined processes per application, so by dynamically creating processes per transaction, at any point in time there can be 240 - 390 single-threaded processes, all of which have work to do. More importantly, the installed CPU capacity (and costs) can be lower, and you are maximizing the usage of those CPU resources. In this example, the maximum workload consumption is 39,000 CPU units, whose installed CPU capacity is 42,000 CPU units (30% less compared to the 60,000 installed CPU units in the other environment). The efficiency is improved because this environment runs 57 - 93% CPU busy (compared to only 40 - 65% CPU busy). The result is less CPU capacity to buy and more efficient use of that capacity.

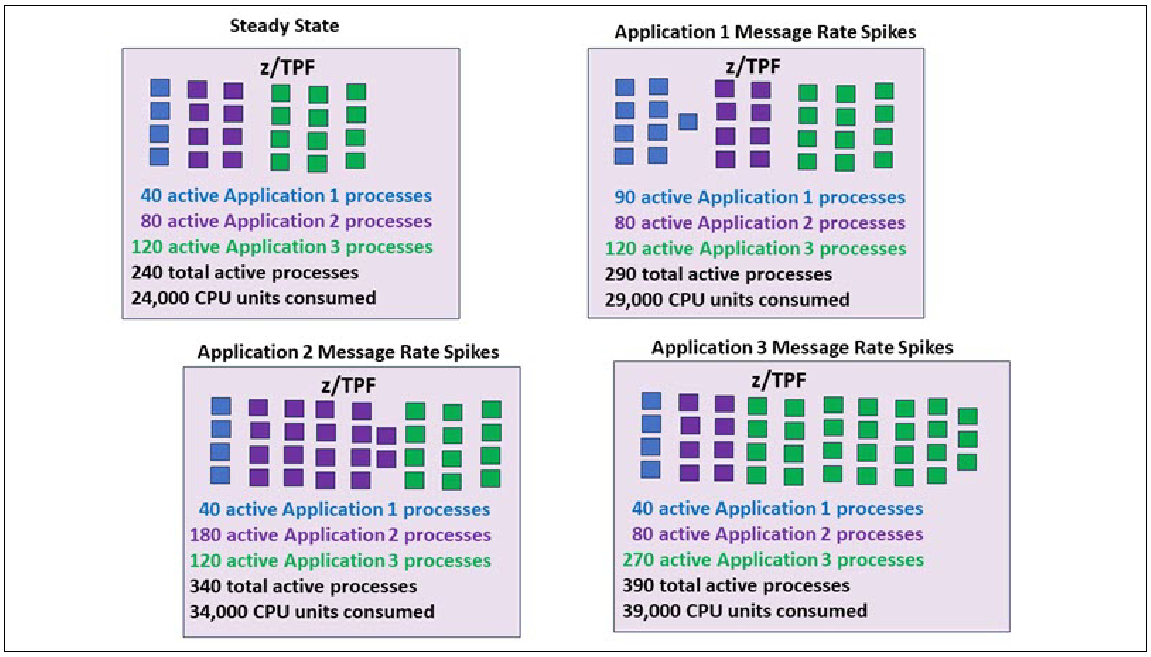

Steady State
z/TPF
40 active Application 1 processes
80 active Application 2 processes
120 active Application 3 processes
240 total active processes
24,000 CPU units consumed
Application 1 Message Rate Spikes
z/TPF
90 active Application 1 processes
80 active Application 2 processes
120 active Application 3 processes
290 total active processes
29,000 CPU units consumed
Application 2 Message Rate Spikes
z/TPF
40 active Application 1 processes
180 active Application 2 processes
120 active Application 3 processes
340 total active processes
34,000 CPU units consumed
Application 3 Message Rate Spikes
z/TPF
40 active Application 1 processes
80 active Application 2 processes
270 active Application 3 processes
390 total active processes
39,000 CPU units consumed

Figure 65 CPU consumption on z/TPF

The next advantage of the z/TPF model is that there is no need for external platform hardware and software to start, restart, and tear down any long-running processes on z/TPF. Monitoring architectures like Kubernetes became necessary for large environments with hundreds of servers, like the upper half of Figure 63 on page 66, where you must detect and restart a failed application process or server node, and activate the application processes on more server nodes if the current environment is saturated and cannot handle the workload. The z/TPF architecture does not use long-running application processes where you must worry about how many processes are needed per application or how to tune the number of threads in each process. Instead, each transaction runs in a new, single-threaded process that handles that one transaction, and then exits.

If there are 200 transactions being processed by application 1 and 500 by application 2 concurrently, there are 700 active single-threaded processes. If at a different point in time there are 400 transactions being processed by application 1 and 300 by application 2, there also are 700 active single threaded processes. In other words, z/TPF can automatically shift resources to whatever application needs it. Rather than monitoring individual applications to see whether each one has enough resources, and monitoring the system itself, you need to monitor only the system to make sure that there are resources to handle the current workloads for all applications, regardless of what the message mix is then. If a single-threaded application process fails, only that one transaction is impacted (there is no need for a monitor to restart anything, and when the next transaction arrives, z/TPF creates a single-threaded application process for that transaction.

The z/TPF database architecture

One key reason why clients choose z/TPF is its database capabilities, which are secure, centralized, highly available, and able to update millions of database records per second in a strict consistent manner. This section explains in detail the advantages of the z/TPF database architecture.

For high availability of data, the z/TPF system keeps two local copies of each database record on separate physical disks that are accessed through two different DASD control units (CUs) so that there is no single point of failure. If a disk or even an entire DASD CU fails, at least one copy of all z/TPF database records are still accessible, and transactions continue processing on that z/TPF system. Figure 66 shows an example of a z/TPF system that is connected to four DASD CUs, with two disks that are attached to each DASD CU.

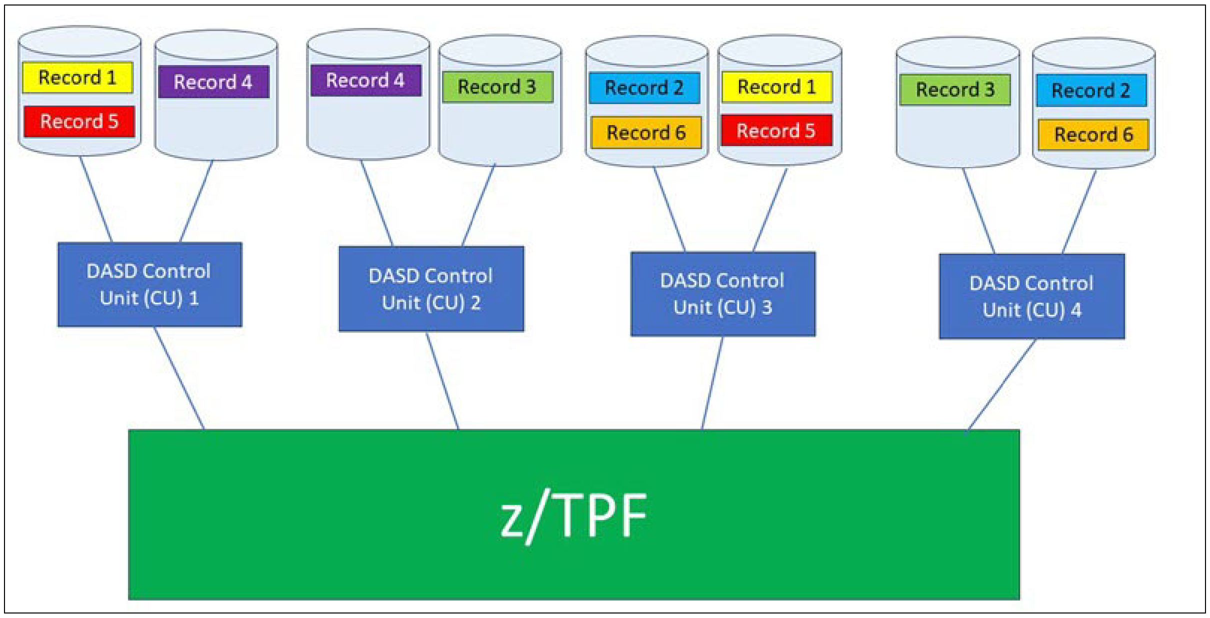


Figure 66 z/TPF duplicated database architecture

On an actual system, there would be hundreds or even thousands of disks that are attached to each DASD CU, but two disks are used in this example to keep the diagram readable. There are six records in the database, with two copies of each record. Whenever z/TPF must read a database record, it can read from either copy on disk. Whenever z/TPF must update and file a record, it writes to both copies on disk in parallel for high performance. For example, if z/TPF wants to read database record 1, it can send the request to DASD CU 1 or DASD CU 3. If z/TPF wants to update database record 1, it sends the updated data to both DASD CU 1 and DASD CU 3 concurrently.

The z/TPF database is horizontally striped, which means records within any local database are spread out across all the disks. For example, you have 1000 disks that are attached to z/TPF and three databases: flight inventory (INV), reservation records (PNR), and flight operations (FOS). If all INV records are assigned to disks 1 - 100, all PNR records to disks 101 - 800, and all FOS records to disks 801 - 1000, and most the database operations are against the INV records, those 100 disks (or the DASD CU that owns those 100 disks) can be overloaded (database hot spots), which impacts the workload response time and ability to scale.

Instead, z/TPF spreads INV records, PNR records, and FOS records across all 1000 disks so that the DASD I/O activity is spread evenly across the disks (and DASD CUs) regardless of which databases are most heavily accessed. Horizontal striping of data also performs better for applications and utilities that step through an entire database (read record 1, then record 2, then record 3, and so on). Using Figure 66 on page 68 as an example, if records 1 - 4 are all on the same disk (vertical striping), then you must read those records serially. Because of z/TPF horizontal data striping, all four records can be read in parallel because they are on different disks, which result in overall better (lower) time to read a set of records.

A major advantage of the z/TPF database is how much faster a z/TPF application can read and update a database record compared to a distributed application when high availability of data is required, which is a must for enterprise workloads.

Figure 67 is a distributed environment with many application servers (only one is shown) and three database servers to handle the workload volume.

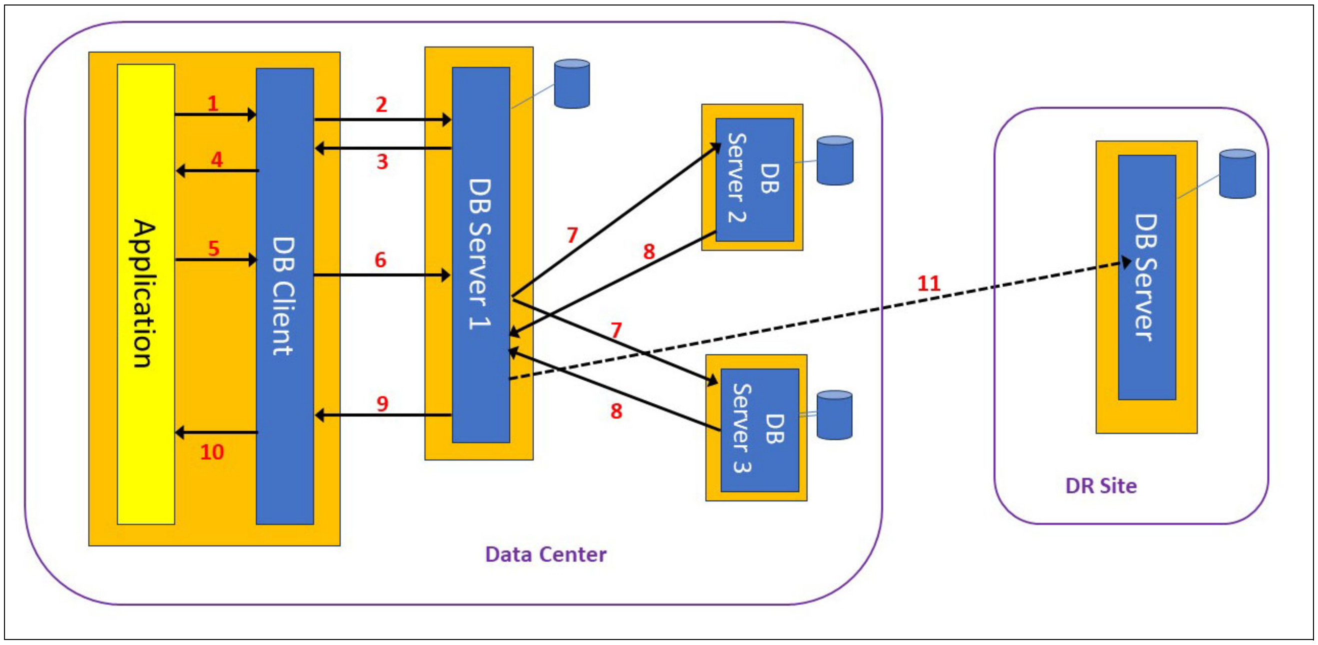


Figure 67 Reading and updating a database record in a distributed architecture

All three database servers maintain copies of the same data. Whenever any record is updated in one database server, the update is propagated to the other database servers and must be acknowledged before the update operation is considered complete to maintain strict consistent data.

Here are the process steps:

1. The application wants to read database record 1, so it calls the database client layer within that application server node.
2. The database client selects a database server (DB server 1 in this example) and sends the request to that server over the network.
3. DB server 1 reads record 1 from its copy of the database and sends it back to the application server node over the network.
4. The contents of record 1 are given to the application.
5. The application changes record 1 and calls the database client layer to file out those changes.
6. The database client sends the update request to DB server 1 over the network.
7. DB server 1 updates the copy of record 1 in its database and sends the update request to the other database servers (DB server 2 and DB server 3) over the network.
8. DB servers 2 and 3 each update the copy of record 1 in their databases and respond to DB server 1 over the network, which indicates that the update was successful.
9. DB server 1 responds to the application server node over the network that the update operation is done.
10. The DB client returns control to the application to indicate that the update request was successful.
11. Asynchronously, DB server 1 sends a copy of the updated record 1 to a database server in the DR site to update the DR copy of the database.

Figure 68 shows the same application reading and updating a database record example, except this time, the application is running on z/TPF and the record is in the z/TPF database.

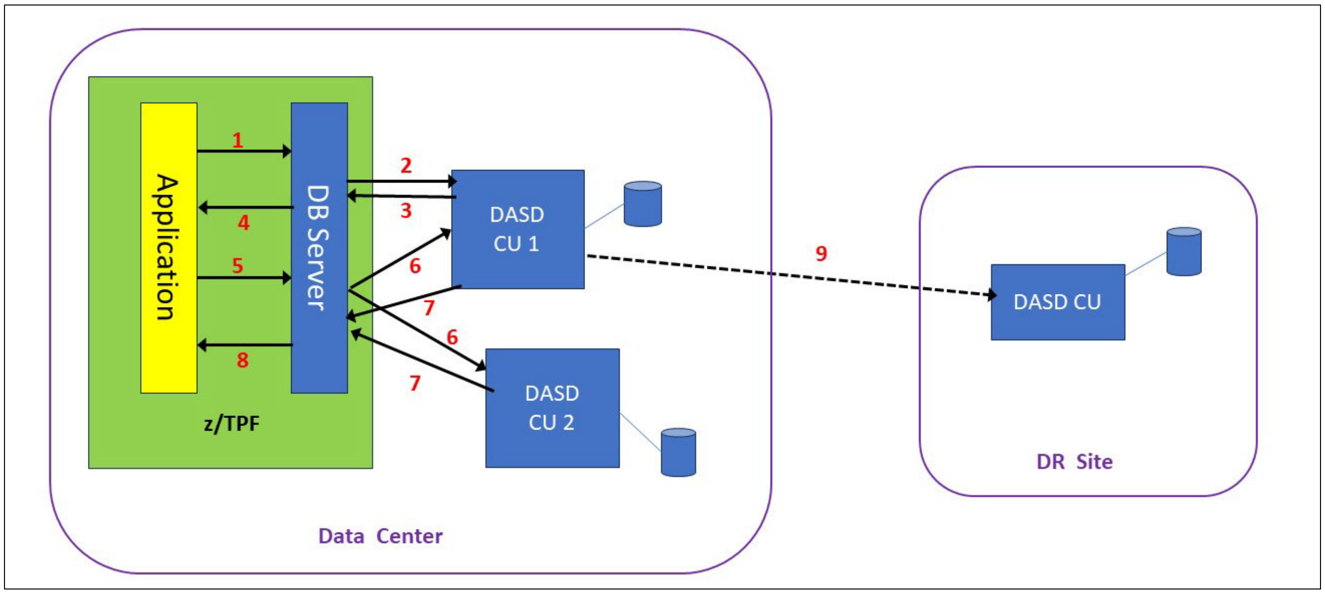


Figure 68 Reading and updating a database record in z/TPF

Here are the steps for this process:

1. The application wants to read database record 1, so it makes a local intra-process call to the z/TPF database server layer.
2. Record 1 exists on a disk that is attached to DASD CU 1 and on a disk that is attached to DASD CU 2, so the database server layer selects which copy of the record to read. In this example, DASD CU 1 is selected and the read request is sent to DASD CU 1.
3. DASD CU 1 reads its copy of record 1 and sends it back to z/TPF.
4. The contents of record 1 are given to the application.
5. The application changes record 1 and calls the z/TPF database server layer to file out those changes.
6. z/TPF sends to the update request in parallel to both DASD CU 1 and DASD CU 2.
7. DASD CU 1 and DASD CU 2 each update their copy of record 1 and respond to z/TPF to indicate that the update was successful.
8. The DB server layer returns control to the application to indicate that the update request was successful.
9. Asynchronously, DASD CU 1 sends a copy of the updated record 1 to a DASD CU in the DR site to update the DR copy of the database.

Both the read request and update request complete in a shorter amount of time by using z/TPF. In the distributed model, reading data requires external network flows between the application server and database server, and in z/TPF, the application and database server are both inside z/TPF and running in the same process. The update request highlights the performance advantages of the z/TPF architecture even more. In the distributed model, there are the external network flows between the application server and database server, but there is also serial processing where the update request first must go to one database server, then to all the other database servers, then back to the original database server, and then finally back to the application server. In z/TPF, both copies of the database are updated concurrently, so the update operation completes faster.

This example of reading and updating a database record shows some advantages of the z/TPF database architecture even when no database record locking is used. Often, locking is needed to serialize updates to the same database record and prevent some updates from being lost. In those cases, the application does a lock and read record operation and then eventually a file and unlock operation, which means that the record is locked from the time it was read until after the updated record is filed. To maintain data consistency in the distributed environment, the lock and read request requires coordination and agreement by all the database servers, which means additional flows between the database servers, so it takes longer to process the application's read request.

Another advantage of the z/TPF database is that it is centralized and contains multiple, unpartitioned logical databases (such as inventory, reservation records, and customer profiles). To understand why this approach is valuable, take transactions that must update multiple database records in a strict consistent manner, which means that all the database updates must be made or none of the database updates are made. A simple example is a transaction that transfers $400 from user X's account (the current balance is $1000) to user Y's account (the current balance is $300). That transaction must update the database account record for user X and the database account record for user Y. If the application server attempts to update both records but the application server, a database server, or both fail, there are four possibilities for the state of the database:

1. Both records were successfully updated, so user X's balance is now $600 and user Y's balance is now $700.
2. Neither record was updated, so user X's balance is still $1000 and user Y's balance is still $300.
3. Only user X's record was updated, so user X's balance is now $600, but user Y's balance is still $300.
4. Only user Y's record was updated, s user X's balance is still $1000, but user Y's balance is now $700.

Possibility 1 is the normal case, which is a successful transaction. Possibility 2 is also OK from a data integrity point of view because the result is that the database was not changed. Possibilities 3 and 4 are unacceptable because the database is in an inconsistent state.

To prevent database inconsistencies like possibilities 3 and 4, applications use transaction scopes (commit scopes). The application opens a transaction scope, reads and updates multiple database records from its point of view (but the updates are not made to the databases yet), and then the application commits the transaction scope, at which time the system makes the database updates and ensures that all updates or no updates are made to the databases.

The money transfer example involved updating two records in the same database. Transaction scopes are also needed when updating records in different databases, for example, moving two seats on flight 1234 from the inventory database into a record in the reservations database.

We now compare the performance of using transaction scopes in a distributed environment versus z/TPF. Figure 69 and Figure 70 show the flows of committing a transaction scope in the distributed environment. In this example, there are two database clusters.

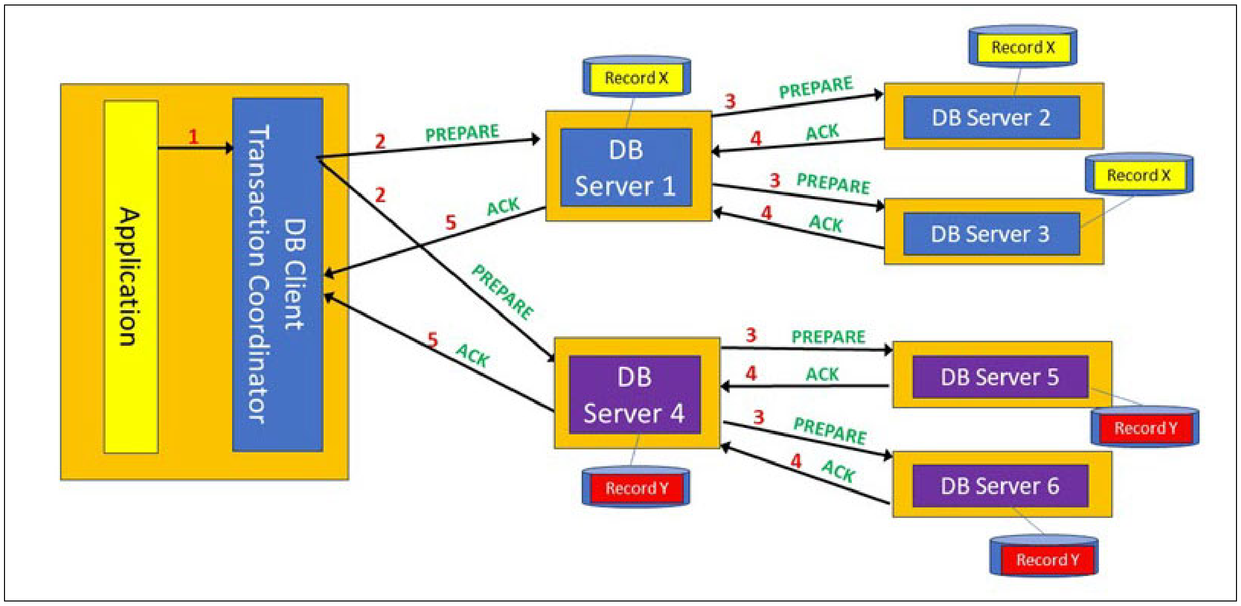


Figure 69 Two-phase commit PREPARE stage

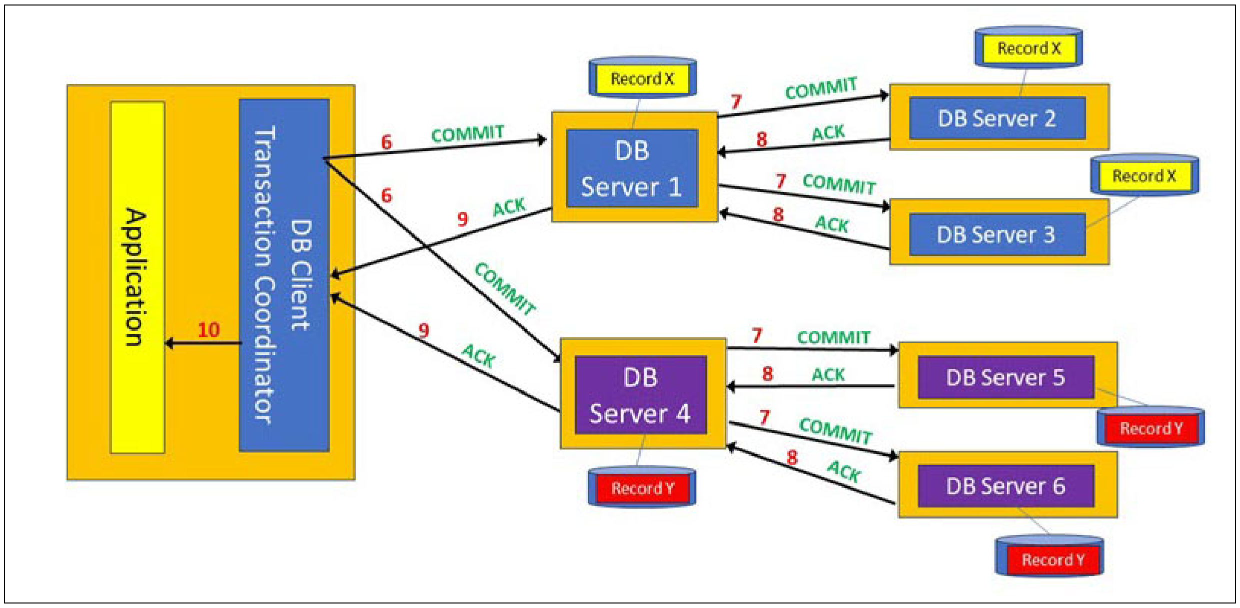


Figure 70 Two-phase commit COMMIT stage

Database servers 1, 2, and 3 represent one cluster and contain user X's account data (Record X). Database servers 4, 5, and 6 represent one cluster and contain user Y's account data (Record Y). Not shown in either of the figures is that the application opened a transaction scope, read records X and Y, and then updated records X and Y, where the updates were written to a local log (so that if the application server fails in the middle of commit processing, a recovery action can be taken when the application server comes back up).

Here are the subsequent steps:

1. The application issues an API to commit the transaction scope.
2. The database client in the application server is the transaction coordinator, which means that it is responsible for communicating with the different database servers to ensure that all or no updates occur by using the standard 2-phase commit process. The first part of the process is to tell all the database servers to PREPARE. The PREPARE message is sent to one database server in each cluster (DB servers 1 and 4 in this example).
3. Within each cluster, the PREPARE message is forwarded to the other servers in that cluster. In this example, DB server 1 does its own preparation, and then forwards the PREPARE message to DB servers 2 and 3. DB server 4 does its own preparation, and then forwards the PREPARE message to DB servers 5 and 6.
4. In cluster 1, the other database servers (DB servers 2 and 3) do their preparation and acknowledge (ACK) that fact to DB server 1. In cluster 2, the other database servers (DB servers 5 and 6) do their preparation and acknowledge (ACK) that fact to DB server 2.
5. The database servers (DB servers 1 and 4) acknowledge to the application server that the PREPARE step is done.
6. The database client in the application server is the transaction coordinator and continues the 2-phase commit process by telling the database servers to COMMIT. The COMMIT message is sent to one database server in each cluster (DB servers 1 and 4 in this example).
7. Within each cluster, the COMMIT message is forwarded to the other servers in that cluster. In this example, DB server 1 updates its copy of record X, and then forwards the COMMIT message to DB servers 2 and 3. DB server 4 updates its copy of record Y, and then forwards the COMMIT message to DB servers 5 and 6.
8. In cluster 1, the other database servers (DB servers 2 and 3) update their copy of record X and acknowledge (ACK) that fact to DB server 1. In cluster 2, the other database servers (DB servers 5 and 6) update their copy of record Y, and then acknowledge (ACK) that fact to DB server 2.
9. The database servers (DB servers 1 and 4) acknowledge to the application server that the COMMIT step is done.
10. Control is returned to the application, which indicates that the commit was successful, meaning that all database updates have been made.

Figure 71 shows the flows of committing a transaction scope in z/TPF. All the databases and database records are in z/TPF.

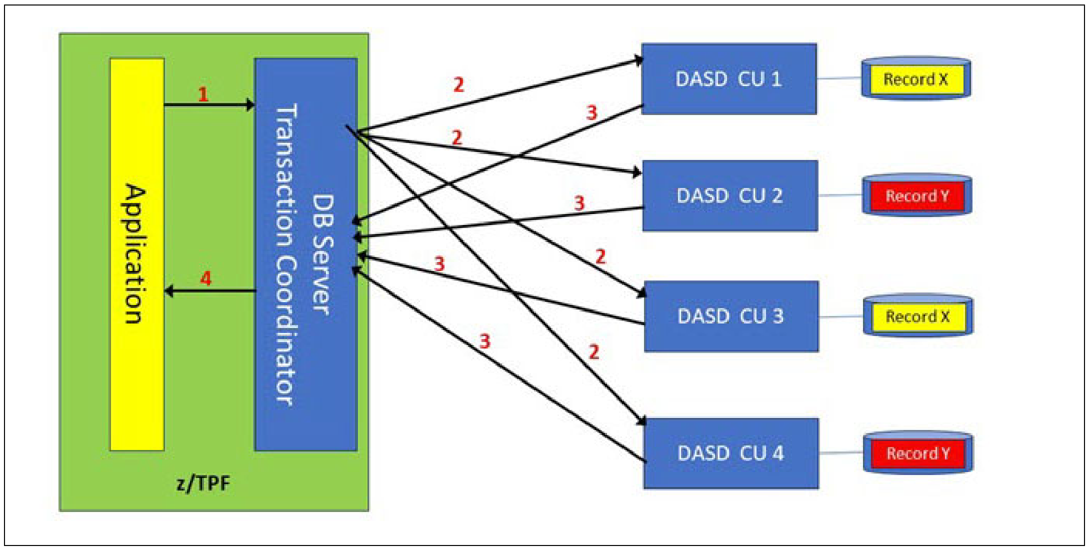


Figure 71 Updating multiple database records in z/TPF

In Figure 71, user X's account data (Record X) and user Y's account data (Record Y) exist in the z/TPF database. Not illustrated in the figure, the application opened a transaction scope, read records X and Y, and then updated records X and Y, where the updates were written to a local log (so that if z/TPF fails in the middle of commit processing, a recovery action is taken where z/TPF comes back up to continue commit processing). Then, the following steps are processed:

1. The application issues an API to commit the transaction scope.
2. The z/TPF database server is the transaction coordinator, which means that it is responsible to ensure that all or no updates occur. z/TPF sends the updated data for all database records to both places in the z/TPF database where those records exist. In this example, the updated record X data is sent to DASD CUs 1 and 3, and the updated record Y data is sent to DASD CUs 2 and 4.
3. Each DASD CU applies the database updates, and then replies to z/TPF to indicate that the database records were updated.
4. Control is returned to the application, which indicates that the commit was successful and all database updates were made.

Committing multiple database updates by using the 2-phase commit process in a distributed architecture is expensive in terms of transaction latency because there are four sets of flows that must be done serially, but in z/TPF there is only a single flow exchange. Some distributed architectures even use a 3-phase commit scope, meaning more flows and even higher latency. Reduced transaction latency is not the only advantage of z/TPF because reduced database lock time, meaning that how long a database record lock is held, is critically important in scaling a workload up to higher transaction volumes. The longer a database lock is held, the higher the likelihood for lock contention, which means another application process is trying to lock that database record but must wait because that lock is held by another application process. High database lock contention is one of the biggest inhibitors for a computer architecture being able to handle higher rates of transactions.

Security is of utmost importance on z/TPF, which includes protecting data in flight, at rest, and in use. In this section, we cover the major advantages of z/TPF in protecting data at rest:

- Entire z/TPF databases can be protected (encrypted), which requires no application changes or outages.
- Data is protected not only while at rest on disk, but it is encrypted at every exposure point between the z/TPF database access layer and the disk.

- Quantum-safe cryptography protects the data from current and future attacks.

In the past, when hardware-accelerated cryptography was limited or not present, it was practical to encrypt a small subset of data in the database. For example, in an airline reservation record, only the credit card number that was used to purchase the trip was encrypted. As more regulations were written, it became necessary to encrypt more information in the database records, such as PII data. Application programs had to know what fields were encrypted versus the fields that were in the clear, and whenever a new regulation was written, application changes were required. Because some data was encrypted and some was not, compliance audits were often tedious to prove to the auditors that all the data that needed to be protected was protected.

The rate and pace of new regulations continues to accelerate. However, due to the vast improvements in hardware cryptography on IBM Z, there is now enough hardware crypto capacity to encrypt every record in every z/TPFDF database on z/TPF. For databases that you define as requiring encryption, the z/TPFDF automatic database encryption support encrypts all the data in those databases, but applications continue to see and work with data in the clear. The fact that data at rest is encrypted is transparent to the applications. This approach also simplifies the audit process because instead of having to prove that PCI data is protected, and then prove that PII data is protected, and so on, for one regulation at a time, you can prove that all data is protected. Simple displays show what databases are protected and by what cryptographic algorithm to prove that an approved algorithm is used.

Years ago, the focus on protecting data at rest meant data that was on disk. Some customers used encrypted disk technology, which meant that the disk or DASD controller encrypted the data before writing the data to disk. In recent years, the definition of “at rest” has expanded sometimes to include data that is cached in memory for more than a certain amount of time. For example, if your z/TPF system has been running for months and all one million records for a database containing sensitive information are in the VFA database memory cache on z/TPF, that might be considered data at rest, and encrypted disk technology does not protect that copy of the data.

Figure 72 shows that data is encrypted at the z/TPFDF database access layer, so the data “at rest” when cached in memory in VFA is encrypted. When z/TPF writes the (encrypted) data to the DASD CU, the DASD CU might keep a copy of the data in its own memory cache, but that copy also is protected (encrypted). z/TPFDF automatic database encryption support ensures that every place an “at rest” cached copy of data might exist will be encrypted data, and also protects the data as it flows between the IBM Z server and DASD CU, the flows between DASD CU in the primary data center and DR site, and at rest in the DR site.

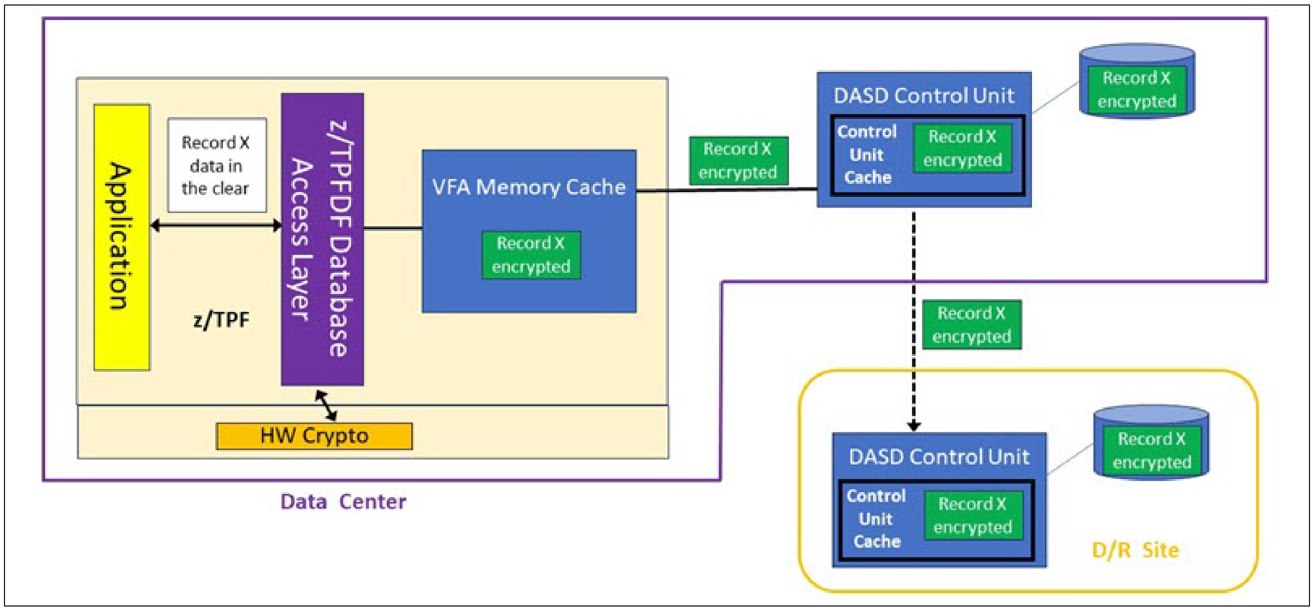


Figure 72   *z/TPFDF Automatic Database Encryption*

In the future, it is theorized that quantum computers will be able to break many cryptographic algorithms that are being used today. Bad actors are employing a “harvest data now, decrypt later” approach. To protect your data today and to combat future threats, you can use quantum safe algorithms today like AES-256 to encrypt your z/TPF databases.

In summary, the z/TPF database architecture has advantages in response time, scale, reduced lock contention, CPU costs, and security.

Large memory exploitation

With IBM z16™, up to 40 TB of memory can be leveraged to reduce CPU costs, lower transaction latency, and allow you to build smarter applications. This section explains in detail the advantages of the z/TPF memory model to achieve the benefits of using large memory and to improve performance.

The z/TPF system caches database records in memory in a system table that is called Virtual File Access (VFA). For many databases, if your IBM Z server has enough memory, you can cache entire databases in memory by using VFA. For example, if you had a customer database with 100 million entries (database records) and each entry was 4,000 bytes of data, you can cache that entire database in memory with a VFA size of less than half of 1 TB (around 400 GB). If you do not have a large VFA, 80 - 90% of the latency when processing a transaction on z/TPF consists of waiting for DASD I/Os operations (reading data from disk) or waiting for responses to requests that are sent to remote systems. To illustrate the benefits of having data cached in memory, take the profile where processing a transaction reads 20 database records, one at a time (read record 123, do some processing to determine what record to read next, read record 713, and so on). If a database record is not cached in memory, a read I/O operation to disk is done and takes 300 microseconds to complete. The application starts processing, suspended while waiting for I/O, continues processing, suspended for I/O again, and so on. If you add up all the time that the application was actively running (not counting the time that it was suspended), the total is 1 ms.

- Small memory cache: Only two of the 20 records that you read were in the memory cache. The transaction response time is 18 read I/Os (18 * 300 microseconds) + application execution (1 millisecond) = 6.4 ms.
- Medium memory cache: Only eight of the 20 records that you read were in the memory cache. The transaction response time is 12 read I/Os (12 * 300 microseconds) + application execution (1 millisecond) = 4.6 ms.
- Large memory cache: All 20 records that you read were in the memory cache. The transaction response time is the application execution = 1.0 ms.

In this example, if your memory cache is large enough to cache entire databases, the transaction response time is reduced by 78 - 84%. There also is a reduction in CPU consumption because doing a read I/O operation from disk consumes more CPU than reading the record from a memory cache. On systems that do hundreds of thousands or millions of database record accesses per second, eliminating read I/Os from disk can result in noticeable CPU savings. The amount of CPU that is saved depends on what percentage of the application run time is spent in the database layer processing read record requests.

Applications on systems that do not have a large database memory cache are often limited to the amount of data that they can access while processing a transaction and still achieve response time SLAs. For example, if your response time SLA states that the application must respond within 10 ms, the application might be designed to access at most 20 - 25 database records because of all the I/O latency. Contrast that response time with an application running on a system with entire databases that are cached in memory that can access hundreds or even thousands of database records and still meet that 10 ms response time SLA. The more data that an application can access in real time, the smarter decisions it can make.

Beyond database records, there is other information that can be cached in memory to reduce CPU consumption and latency. A prime example is caching results data: If your applications are asked repeatedly the same questions that require a significant amount of CPU and I/O resources to compute the answer, cache the results of that query.

For example, Figure 73 shows that your flight availability application receives 20 messages per second about what flight options there are for going from Boston to Denver on Nov 8. The answer to that question does not change frequently, so it is not cost-effective to compute the same answer 20 times per second.

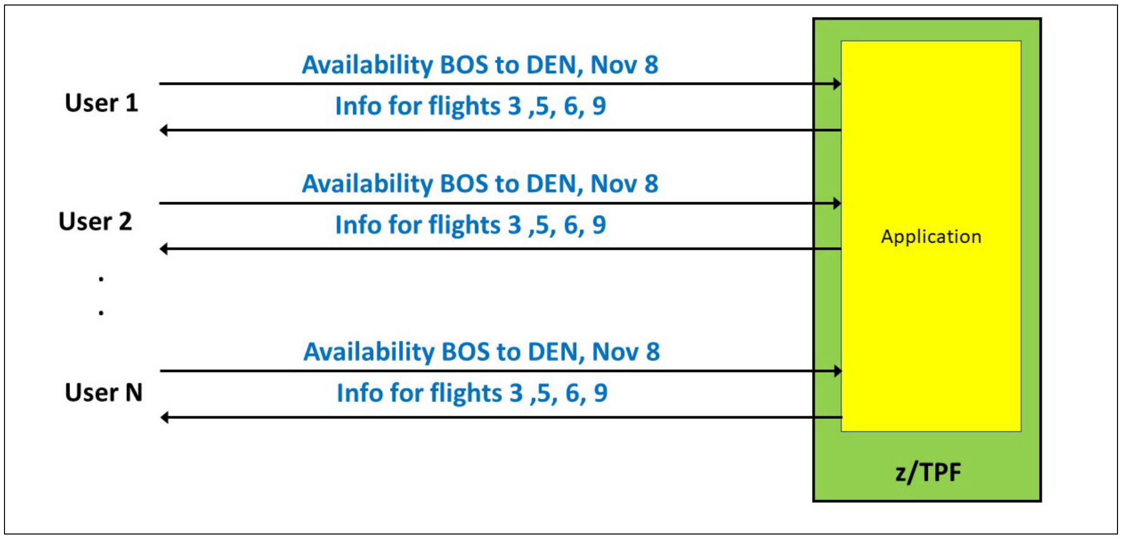

User 1
Availability BOS to DEN, Nov 8
Info for flights 3 ,5, 6, 9
User 2
Availability BOS to DEN, Nov 8
Info for flights 3 ,5, 6, 9
User N
Availability BOS to DEN, Nov 8
Info for flights 3 ,5, 6, 9
Application
z/TPF

Figure 73 Flight availability application: Many users requesting the same information

Instead, you can compute the answer once and save the result in a memory cache. When subsequent requests ask the same question, use the already computed result, as shown in Figure 74.

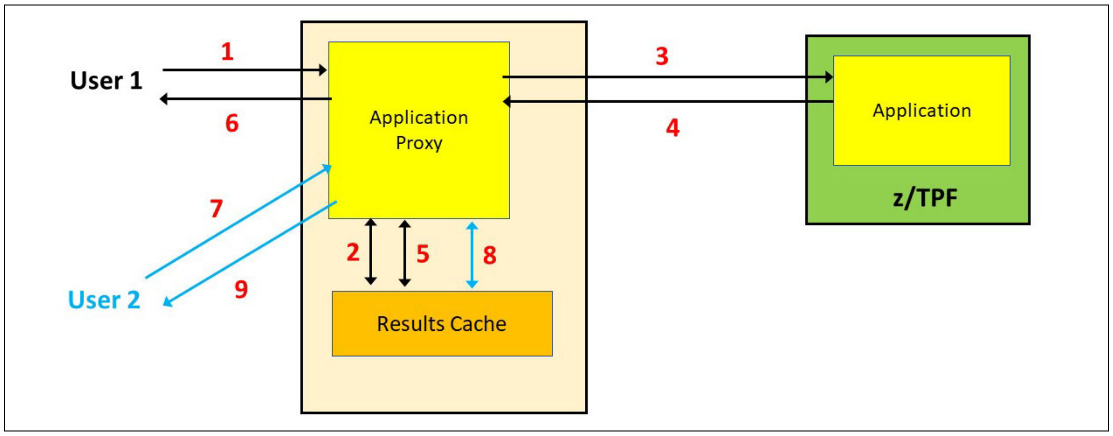


Figure 74 Leveraging external cache

Instead of the user sending the requests to z/TPF, the results are sent to an external application proxy node that contains a results cache. If the application proxy has the answer in its results cache, it responds to the user with that cached result. If the application proxy does not have the information in its cache, it forwards the request to z/TPF, which calculates the answer. Then, the application proxy responds to the user with that information and saves the information in its results cache.

Here are the process steps:

1. User 1 sends a request to the application proxy for flight options BOS to DEN on Nov 8.
2. The application proxy searches its cache to see whether it has the answer to that query. It does not.
3. The application proxy forwards the request to z/TPF.
4. The application on z/TPF calculates the answer and returns the information to the application proxy.
5. The application proxy saves the information in its results cache.

6. The application proxy responds to User 1.
7. User 2 sends a request to the application proxy for flight options BOS to DEN on Nov 8.
8. The application proxy searches its cache to see whether it has the answer to that query. It does.
9. The application proxy responds to User 2 with the cached information.

Steps 7 - 9 are repeated for other users that request this information.

Although result caches have the advantage of reducing CPU consumption, they do come with their own set of issues, the largest of which is that at some point the cached result will no longer be accurate (will become stale data). Therefore, you need some mechanism to invalidate or refresh the cached data. What mechanism that you use depends on the tolerance, if any, for potentially stale data. For example, if users try to book a flight and it fails because you showed them inaccurate flight options (cached stale results), it leads to customer dissatisfaction.

One commonly used method is to allow each cached result to be used only for a certain amount of time, like 1 - 10 seconds. This method is not perfect because there is the possibility that stale results are used. This method can also cause unnecessary overhead by throwing away cached results that are still valid (and spending resources to recalculate the same answer again). At times, a cached result for a specific query might be valid for 60 seconds, but at other times the result is valid for only 10 seconds or 2.5 seconds. Therefore, expiring cached results at fixed intervals have mixed results.

Figure 75 shows an environment that has many application proxy nodes, which might be because of scale and the need to have multiple proxies to handle the large workload, different geographies deploying their own cache, and so on.

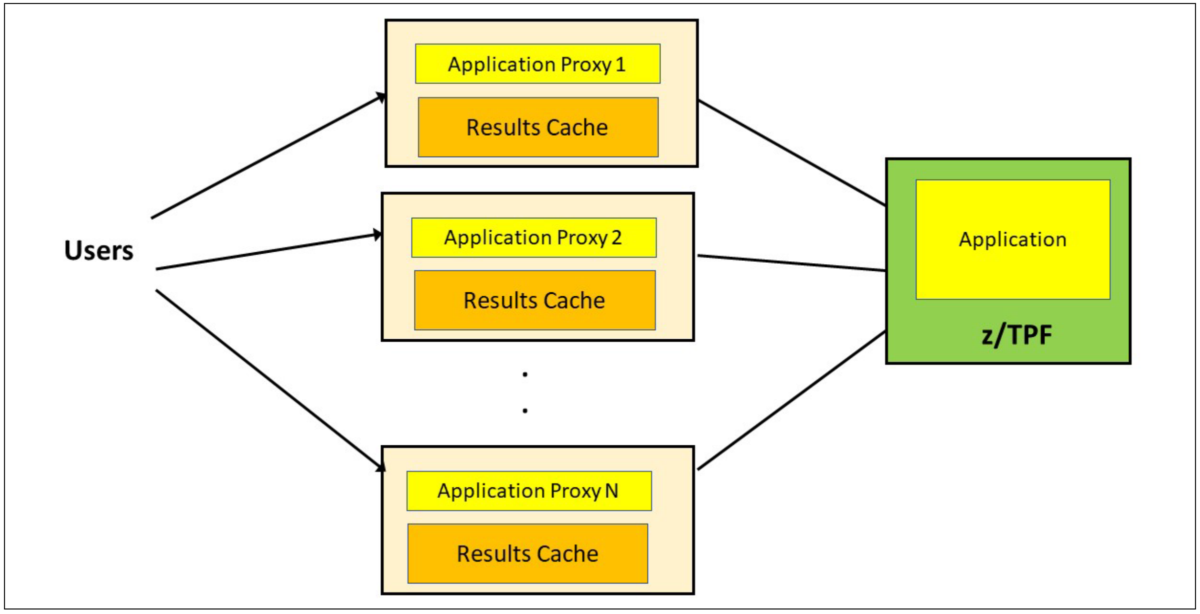
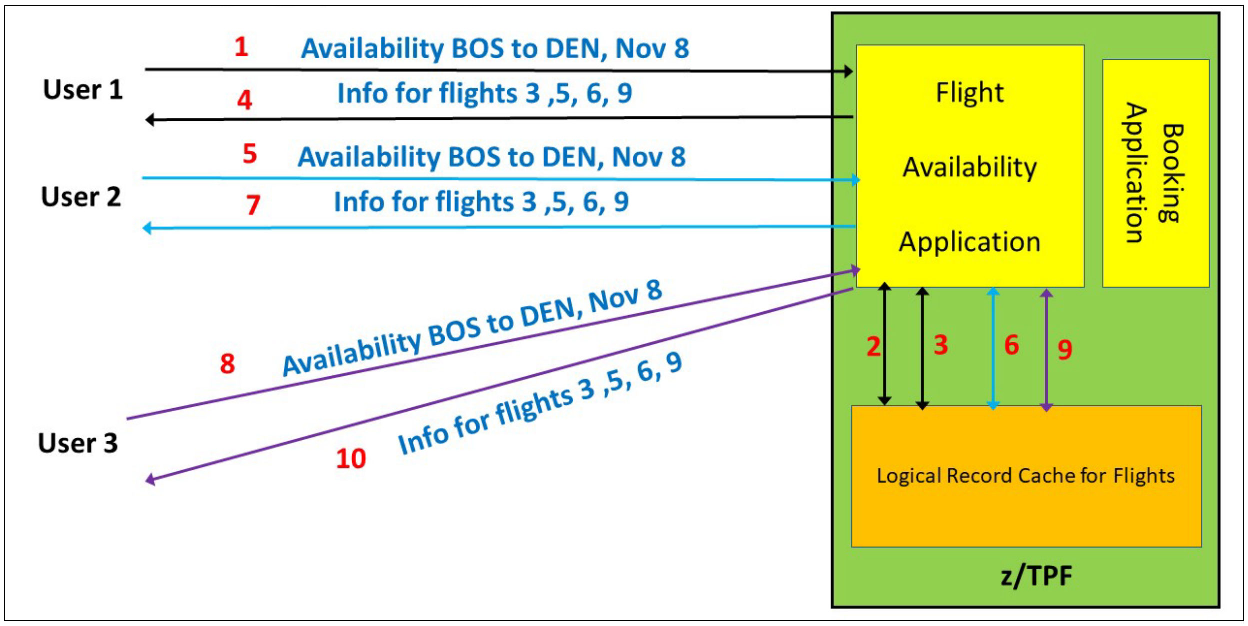


Figure 75 Leveraging external cache at scale

z/TPF Logical Record Cache (LRC) is a results memory cache that is available for use by z/TPF applications. This cache can handle the workload with low CPU consumption and 100% data accuracy (no stale data). Use the flight availability example to explain how this goal can be achieved. The Flight Availability and Booking applications are on z/TPF.

Figure 76 on page 79 shows an example of leveraging z/TPF LRC.



User 1
1 Availability BOS to DEN, Nov 8
4 Info for flights 3 ,5, 6, 9
User 2
5 Availability BOS to DEN, Nov 8
7 Info for flights 3 ,5, 6, 9
User 3
8 Availability BOS to DEN, Nov 8
10 Info for flights 3 ,5, 6, 9
Flight Availability Application
Booking Application
2
3
6
9
Logical Record Cache for Flights
z/TPF

Figure 76 Leveraging z/TPF Logical Record Cache

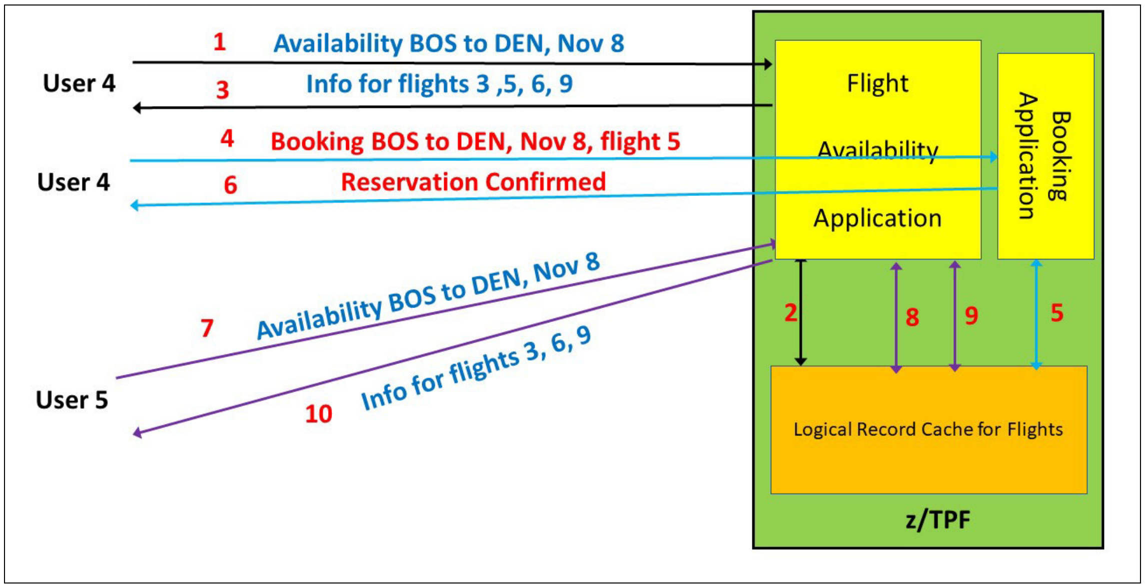
When the Flight Availability application receives a request, it checks the LRC to see whether a cached result exists. If so, the application uses the cache. If not, the application calculates the result and saves the answer in the LRC.

Here are the process steps for this case:

1. User 1 sends a flight availability request to z/TPF for BOS to DEN on Nov 8.
2. The Flight Availability application checks the LRC to see whether an answer exists for this query. It does not.
3. The Flight Availability application calculates the answer (flights 3, 5, 6, and 9) and saves that data in the LRC.
4. The Flight Availability application responds to User 1 that flights 3, 5, 6, and 9 have availability.
5. User 2 sends a flight availability request to z/TPF for BOS to DEN on Nov 8.
6. The Flight Availability application checks the LRC to see whether an answer exists for this query. It does, so it is used.
7. The Flight Availability application responds to User 2 that flights 3, 5, 6, and 9 have availability.
8. User 3 sends a flight availability request to z/TPF for BOS to DEN on Nov 8.
9. The Flight Availability application checks the LRC to see whether an answer exists for this query. It does, so it is used.
10. The Flight Availability application responds to User 3 that flights 3, 5, 6, and 9 have availability.

When the Booking application detects that a flight is sold out, it invalidates (deletes) the corresponding flight availability entry in the LRC. The next flight availability request for that route does not find the answer in the LRC, so it calculates the new answer and saves it in the LRC.

Figure 77 continues the example by using the z/TPF LRC to show how to prevent stale data from being used.



User 4
1 Availability BOS to DEN, Nov 8
3 Info for flights 3 ,5, 6, 9
User 4
4 Booking BOS to DEN, Nov 8, flight 5
6 Reservation Confirmed
User 5
7 Availability BOS to DEN, Nov 8
10 Info for flights 3, 6, 9
Flight Availability Application
Booking Application
2
8
9
5
Logical Record Cache for Flights
z/TPF

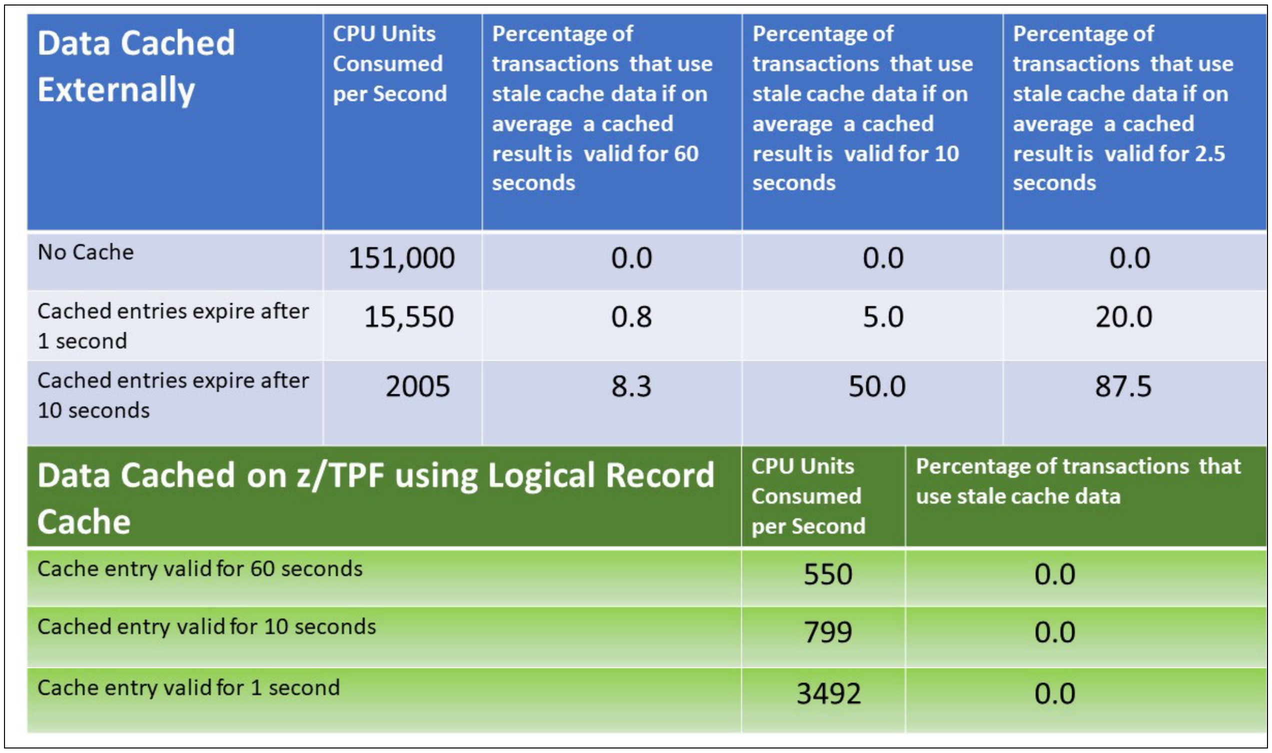
Figure 77 Using z/TPF LRC to prevent stale data

Here are the process steps:

1. User 4 sends a flight availability request to z/TPF for BOS to DEN on Nov 8.
2. The Flight Availability application checks the LRC to see whether an answer exists for this query. It does, so it is used.
3. The Flight Availability application responds to User 4 that flights 3, 5, 6, and 9 have availability.
4. User 4 sends a booking request for flight 5.
5. The Booking application processes the request. The booking is successful, but now flight 5 is sold out. Therefore, the Booking application deletes the LRC cache entry for flight availability for BOS to DEN on Nov 8.
6. The Booking application responds to User 4 that the reservation is confirmed.
7. User 5 sends a flight availability request to z/TPF for BOS to DEN on Nov 8th.
8. The Flight Availability application checks the LRC to see whether an answer exists for this query. It does not.
9. The Flight Availability application calculates the answer (flights 3, 6, and 9) and saves that data in the LRC.
10. The Flight Availability application responds to User 5 that flights 3, 6, and 9 have availability.

Caching the results information in the z/TPF LRC rather than externally gives your applications the ability to keep the cache always accurate with minimal overhead. When a z/TPF application detects a condition that makes a cached LRC result no longer valid, the z/TPF application can issue a local function to immediately invalid (delete) that cached LRC entry from memory on z/TPF.

The top half of Figure 78 shows CPU consumption and cached data accuracy for a few different scenarios when using an external cache.



Data Cached Externally	CPU Units Consumed per Second	Percentage of transactions that use stale cache data if on average a cached result is valid for 60 seconds	Percentage of transactions that use stale cache data if on average a cached result is valid for 10 seconds	Percentage of transactions that use stale cache data if on average a cached result is valid for 2.5 seconds
No Cache	151,000	0.0	0.0	0.0
Cached entries expire after 1 second	15,550	0.8	5.0	20.0
Cached entries expire after 10 seconds	2005	8.3	50.0	87.5

Data Cached on z/TPF using Logical Record Cache	CPU Units Consumed per Second	Percentage of transactions that use stale cache data
Cache entry valid for 60 seconds	550	0.0
Cached entry valid for 10 seconds	799	0.0
Cache entry valid for 1 second	3492	0.0

Figure 78 Comparison of CPU consumption and cached data accuracy

Here are the environment details that are used and measured in the top half of Figure 78:

- There are five external application proxies, each of which gets 10 requests per second for the same query.
- If the application proxy has the answer in its results cache, it costs 10 CPU units to process that request.
- If the application proxy does not have the answer in its results cache, it costs 3020 CPU units to process that request (3000 for z/TPF to calculate the answer, and 20 for the application proxy flows and to add the results to its cache).

You could use no caching, which has the advantage of 100% data accuracy (stale data is never used), but the disadvantage is having the highest CPU costs. You could keep cached results for a long time (10 seconds), which has low CPU costs, but an unacceptably high percentage of stale data being used when the real answer is changing frequently. Even the middle of the road implementation (results are cached for 1 second) can end up with a higher than acceptable amount of stale data at times.

The bottom half of Figure 78 shows the CPU consumption on z/TPF depending on how long a cached result in the LRC remains valid and is used. Here are the environment details that are used and measured:

- If the Flight Availability application has the answer in the LRC, it costs 10 CPU units to process that request (the same cost as an external application proxy example finding the answer in its results cache).
- If the Flight Availability application does not find the answer in the LRC, it costs 3000 CPU units to process that request and save the answer in the LRC (the same z/TPF cost for the example where an external proxy did not have the answer in its cache and had to call z/TPF to calculate the answer).
- If the Booking application detects that a flight is sold out and must delete an entry from the LRC, it costs 2 CP units.

Comparing the charts in Figure 78, they show that using the z/TPF LRC is best practice because it results in lower CPU consumption compared to using an external cache solution, and it always provides an accurate answer.

This example assumes that the external results cache and z/TPF LRC are large enough to hold all the results that you want to cache. If your workload has many results that must be cached, it might not be possible to cache everything in an external cache because of the memory limitations on those servers. An IBM Z server can have terabytes of memory, so an LRC can be hundreds of gigabytes to terabytes, so you can cache all the results that you want to cache.

z/TPF memory and high availability

In addition to leveraging large memory on z/TPF (as described in “Large memory exploitation” on page 76), there also are performance advantages to accessing memory on z/TPF. This section explains in detail the advantages of the z/TPF memory model.

There are performance advantages to accessing any memory on z/TPF. One differentiator is that all memory that z/TPF applications access is permanently backed by real memory. Most operating systems implement memory paging that you can use to allocate more virtual memory to processes than the amount of physical memory that exists.

A disadvantage to this architecture is that if an application program attempts to access an area of virtual memory that currently is not in physical memory, the application process is suspended. The memory access results in disk I/O to page (write out) a section of virtual memory that is in physical memory but has not been accessed in a while, which frees up some physical memory. Then, a read I/O from disk is done that reads the section of virtual memory back into physical memory, and the application process is dispatched. Thus, applications accessing memory might result in loss of control and disk I/O. If this situation happens frequently while processing a transaction, it impacts CPU consumption and transaction response time. z/TPF does not use memory paging at all, so every memory access by a z/TPF application is to real memory, which is fast and never requires I/O.

Another important performance aspect of memory usage on z/TPF is pre-allocated memory. For example, when processing a transaction, your applications on average get and use 100 work areas of memory that total around 4 MB. Some transactions use less than 4 MB, but 98% use less than 6 MB. If you are processing 10,000 transactions per second, on other platforms that results in 1 million get memory calls per second by applications (causing the system to attach the requested memory to that application process) and 1 million release memory calls per second to return the memory to the system. That number of calls to get and release memory can consume many CPU resources.

To avoid all that overhead, z/TPF permanently pre-allocates a user-defined amount of storage to each application process. In this example, the user knows 98% of transactions obtain less than 6 MB of storage, so the user defines the z/TPF pre-allocated memory area as 6 MB, which means that every time an application starts processing a transaction, there is 6 MB of storage that is already attached to the application process that is available for use. When the application issues a get memory call, the system uses the pre-allocated memory, which is faster and has considerably less overhead than having to attach memory to this application process. Only if all the pre-allocated memory is in use does z/TPF attach more memory to the process. In this example, processing 10,000 transactions per second each with 6 MB of pre-allocated memory, the number of times memory must be attached to an application process drops to only 500 per second (compared to 1 million times per second without pre-allocated memory). Properly configured pre-allocated memory on z/TPF reduces CPU consumption.

Kernel-scoped sockets and middleware

On z/TPF, sockets are owned by the kernel and not tied to any application process, which improves performance, scale, and resiliency. This section explains in detail the advantages of this architecture and the z/TPF networking and middleware models.

Starting a new socket by using the TCP protocol uses a 3-way handshake where there is a flow from the client to the server, and then a flow from the server back to the client before the client can send its request message. To avoid the CPU overhead and latency of starting a new TCP connection for every transaction, a common method, which is illustrated in Figure 79, is to reuse the same sockets for multiple transactions. Reusing sockets becomes even more important when there are mandates that secure sockets must be used because starting a secure session first includes the TCP handshake, but then the TLS handshake involves an extra two flows between client and server (extra latency). There can also be considerable CPU overhead starting a TLS session, especially on platforms that do not have hardware-accelerated cryptography.

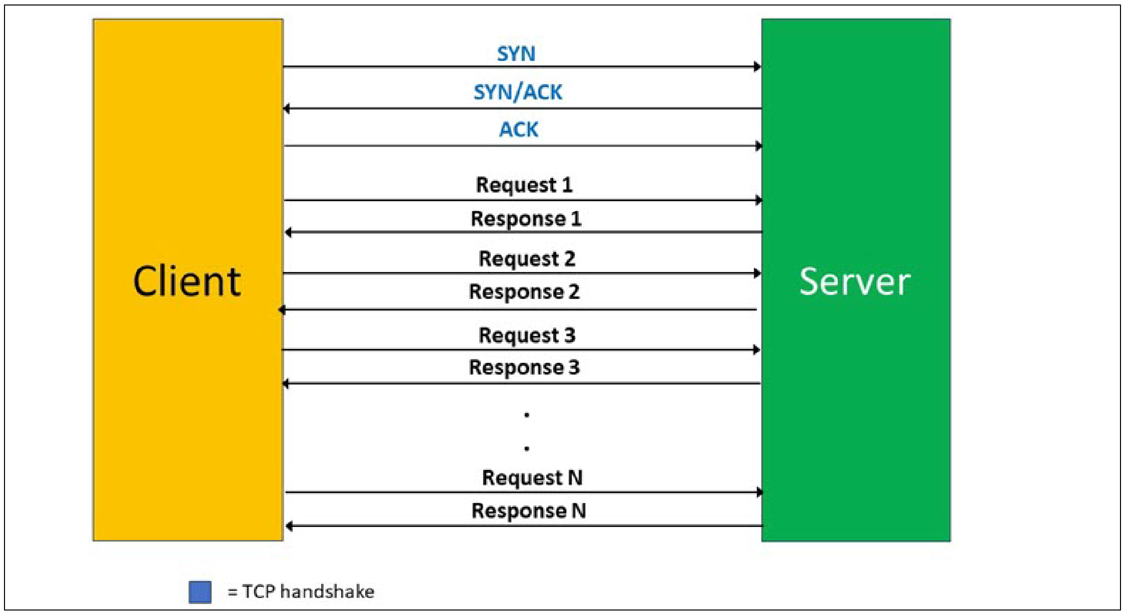


Figure 79 Reusing the same socket for multiple transactions

Figure 80 shows reusing one TLS session for multiple transactions.

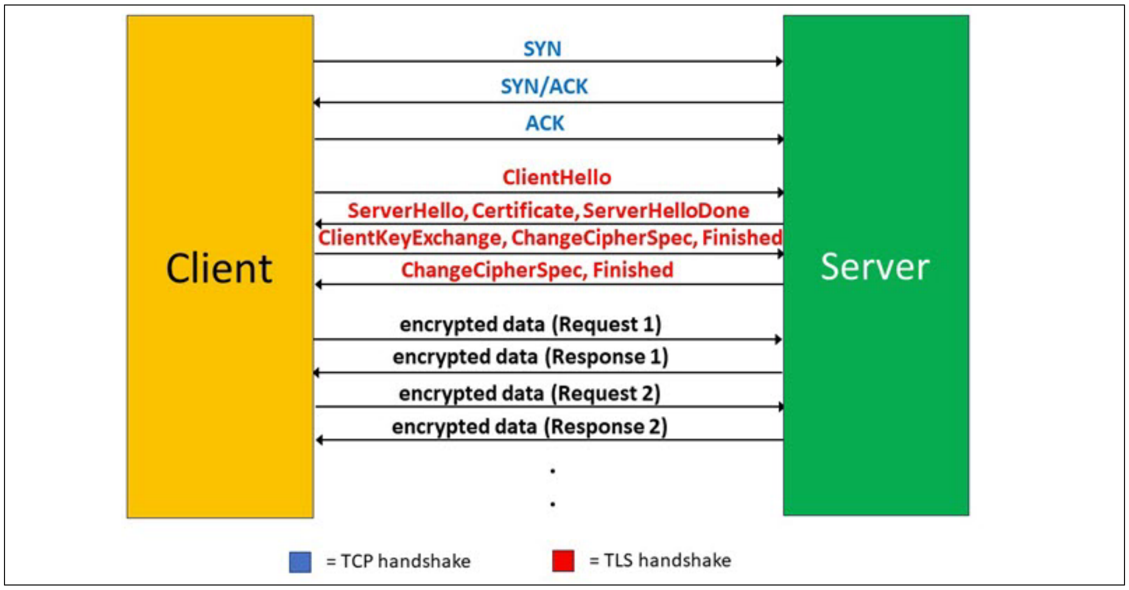

Client
Server
SYN
SYN/ACK
ACK
ClientHello
ServerHello, Certificate, ServerHelloDone
ClientKeyExchange, ChangeCipherSpec, Finished
ChangeCipherSpec, Finished
encrypted data (Request 1)
encrypted data (Response 1)
encrypted data (Request 2)
encrypted data (Response 2)
🟦 = TCP handshake
🟥 = TLS handshake

Figure 80 Reusing one TLS session for multiple transactions

There are thousands of clients that are connected to a server application. You adopt continuous delivery where you load new versions of the server application multiple times per day. On other platforms where the server application is running in a long running process, when you deploy a new version of the application, the existing process that uses the old version of the application is stopped, which causes that process to exit. Then, you restart the application, which causes a new process to be created that starts using the new version of the application. When the old application process exits, the TCP/IP stack breaks all the network connections that are associated with that process, which causes all the clients to reconnect immediately or the next time an individual client has data to send. This process is disruptive to the clients, and can cause a CPU spike on the server with thousands of clients trying to reconnect in a short period of time. In addition to the planned load of a new version of the application, these same problems occur if the TCP/IP stack, middleware, or application fails, which causes the process to be terminated (forced exit).

Figure 81 on page 85 shows an example of loading a new version of an application on other platforms.

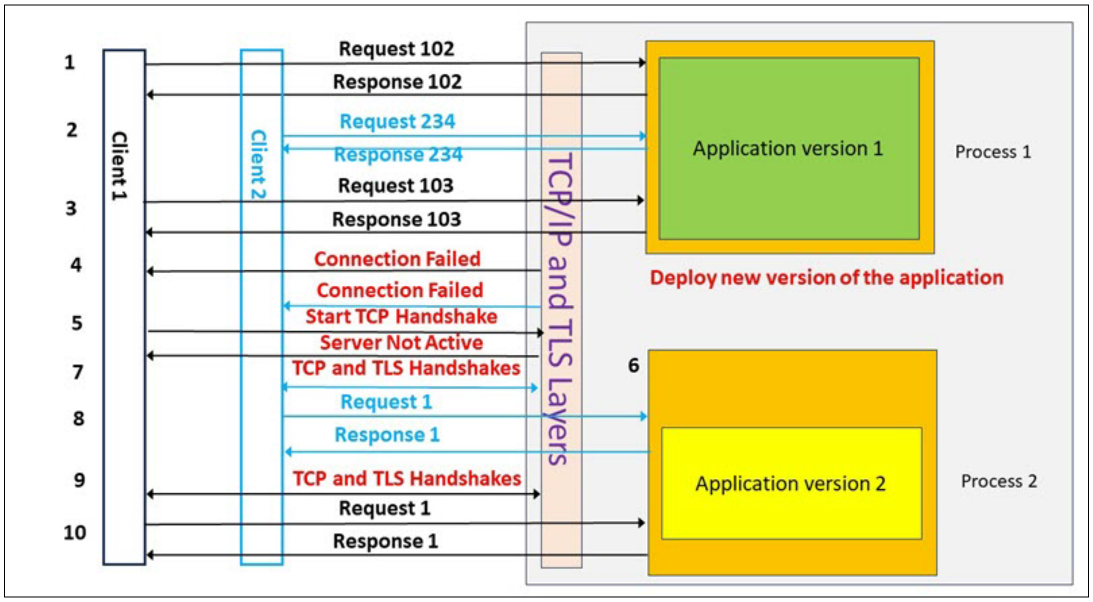


Figure 81 Loading a new version of an application on other platforms

Only two clients are shown in Figure 81, but there could be thousands. Each client has established a secure network connection to the server and so far client 1 has exchanged 101 messages over its connection and client 2 has exchanged 233 messages, all using server application version 1.

Here is the sequence of events from that point:

1. Client 1 exchanges message 102 with the server.
2. Client 2 exchanges message 234 with the server.
3. Client 1 exchanges message 103 with the server.
4. A new version of the server application (version 2) is deployed, so the existing server application stops. Process 1 (where server application version 1 was running) exits, which causes the TCP/IP stack to break the connections with all the clients.
5. Client 1 sees that its existing network connection has failed, so it starts a new connection. However, its TCP handshake is rejected by the server because the server application is not active.
6. The server application is restarted, which creates a new long-running process (Process 2 in this example) where the new version of the server application is now running.
7. Client 2 sees its network connection fail, so starts a new connection with the server, which involves all the TCP and TLS handshake flows.
8. Client 2 exchanges its first message with the server over the new network connection.
9. After waiting for a period of time, client 1 attempts to start a new connection with the server, which is successful this time. This attempt involves all the TCP and TLS handshake flows.
10. Client 1 exchanges its first message with the server over the new network connection.

On z/TPF, you can load a new version of an application nondisruptively, which includes not breaking existing network connections. Figure 82 shows the same example of loading a new version of an application. Only two clients are shown in the diagram, but there could be thousands.

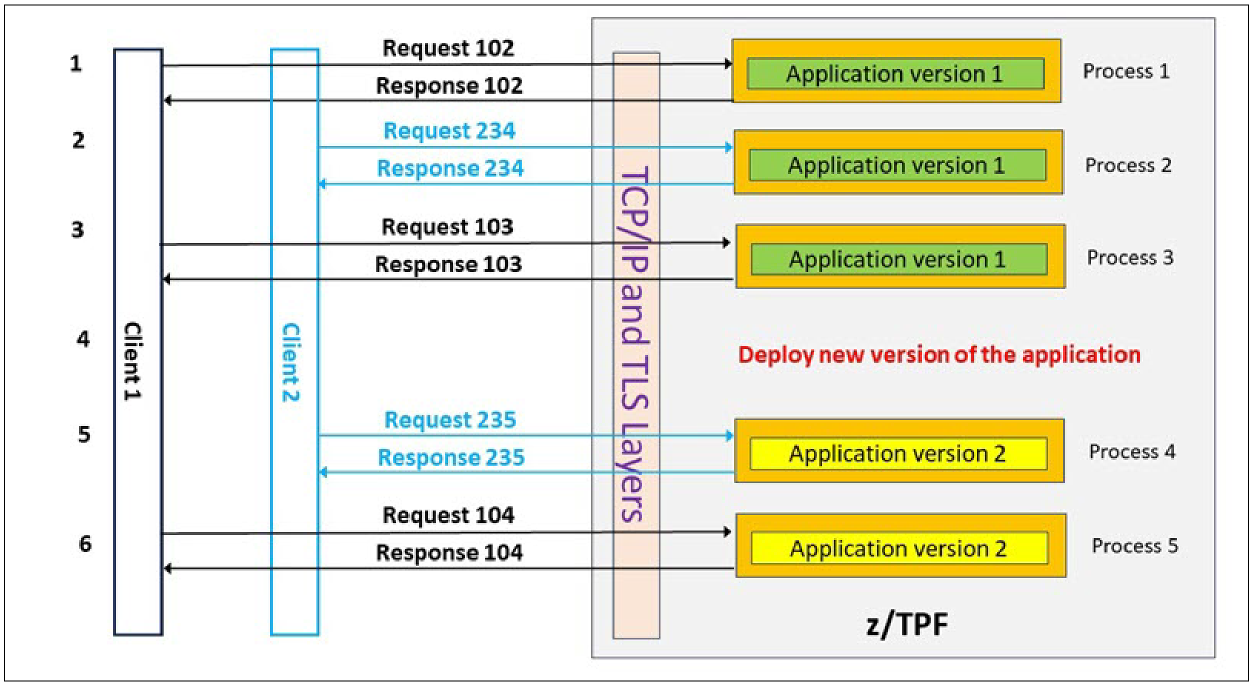


Figure 82 Loading a new version of an application on z/TPF

Each client has established a secure network connection to the server, and so far client 1 has exchanged 101 messages over its connection and client 2 has exchanged 233 messages, all by using server application version 1. Here is the sequence of events from that point:

1. Client 1 exchanges message 102 with the server. The new process 1 is created to handle this one transaction and uses server application version 1.
2. Client 2 exchanges message 234 with the server. The new process 2 is created to handle this one transaction and uses server application version 1.
3. Client 1 exchanges message 103 with the server. The new process 3 is created to handle this one transaction and uses server application version 1.
4. A new version of the server application (version 2) is deployed. Any new messages that arrive at z/TPF use version 2. All network connections remain active.
5. Client 2 exchanges message 235 with the server. The new process 4 is created to handle this one transaction and uses server application version 2.
6. Client 1 exchanges message 104 with the server. The new process 5 is created to handle this one transaction and uses server application version 2.

With the z/TPF model, there is no disruption of traffic when a new version of an application is deployed, and there is no overhead that is required to restart all client connections.

Because of the overhead of starting new network connections, a best practice is for clients to establish connections and leave them active to enable the reuse of connections by multiple transactions, and then enabling the z/TPF socket sweeper to clean up connections that have been inactive for an extended period of time. For example, on z/TPF, you might have 200,000 active connections that generate a total of 2000 messages per second. Many other platforms cannot handle hundreds of thousands of active socket connections because of the number of active processes or threads that would be required, the amount of memory that would be needed for that many connections, or both. On z/TPF, if there were 500 transactions being processed, there would be only 500 (single-threaded) application processes, regardless of whether there were 10,000 active network connections or 500,000 active network connections. Therefore, z/TPF can support hundreds of thousands of network connections efficiently, both in terms of scale and avoiding the overhead of starting thousands of new connections every second.

In a hybrid cloud environment, applications on server B might need to call services of server C 2000 times per second, call services on server D 5000 times per second, and call services on server E 8000 times per second. When servers C, D, and E are behaving normally, they respond in around 1 ms, which means on average at any point in time on server B there are 15 application processes or threads waiting for responses from the remote servers. However, when there are problems with one of those remote servers, it might take 1 - 2 seconds to respond, which can quickly lead to thousands of processes or threads being stuck waiting for responses from one remote server, which can impact all workloads on server B. Because of kernel sockets, one z/TPF application process can receive a request from a remote client and a different application process can send the response to that client over the network connection.

Similarly, when a z/TPF application is acting as a client consuming remote services, the application process that sends the request to a remote server can exit such that there is no active application process while waiting for the response from the remote server. When the response from the remote server arrives, a new application process is created to continue processing the transaction. If one remote server is slow to respond, only the workloads that use that remote service are impacted, and other workloads on z/TPF continue processing transactions normally because there are not thousands of active application processes.

Figure 83 shows a common implementation of a z/TPF application that needs to consume a remote service. Before what is depicted in the diagram, the client established a REST connection with z/TPF and exchanged several REST request and response messages. While waiting for the next request from this client, there is no application process in z/TPF.

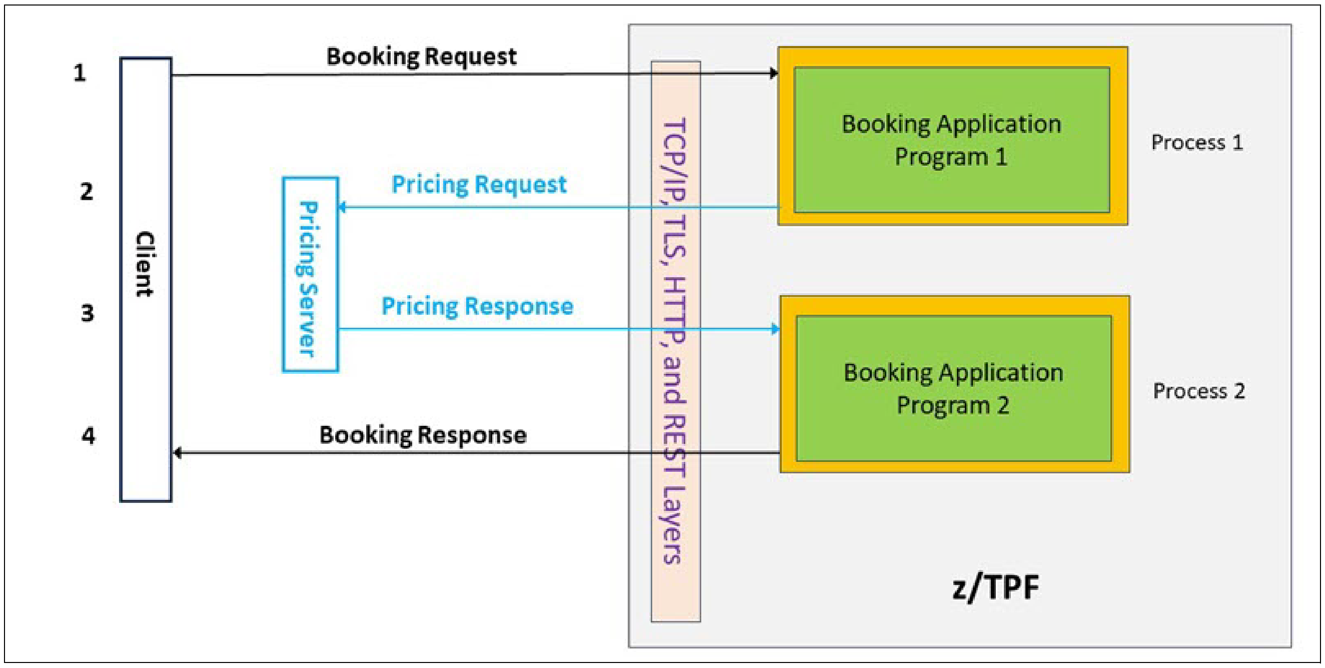


Figure 83 z/TPF application consumes remote service asynchronously

Here are the additional process steps:

1. The client sends a REST request booking message to z/TPF over its existing network connection. This request creates an application process (called process 1) on z/TPF, and booking application program 1 starts processing the transaction.
2. Booking application program 1 makes a REST call to the remote pricing server and tells z/TPF that when the REST response is received from the remote system, pass that response to booking application program 2 in a new application process. Booking application program 1 exits, so process 1 no longer exists. There is no application process waiting for the response from the remote pricing server.
3. The pricing server sends the REST response to z/TPF, which causes z/TPF to create a new process (called process 2) and give the pricing response to booking application program 2.
4. Booking application program 2 finishes processing the transaction and sends the booking response to the client over that client's REST connection. Booking application program 2 exits, so process 2 no longer exists. There is no application process waiting for the next request from the client.

In this example, z/TPF acts as a network server and client, both of which use kernel-scoped network connections to scale up the number of client connections to z/TPF and protect the z/TPF system from one misbehaving remote server impacting workloads that are not using that remote server.

Getting started with IBM

Innovate and co-create with IBM experts to solve your most complex business challenges:

- Accelerate your application modernization journey.
- Achieve meaningful and scalable business outcomes.

Co-creating with IBM

IBM Consulting and IBM Client Engineering deliver meaningful and scalable business outcomes across all industries. With our deeply skilled multi-disciplinary squad and human-centered approach, we provide value-based experiences and solutions that cater to your organization’s needs.

Whether you want a custom demonstration in your environment or an MVP to prove value, we meet you where you are and work with your organization at any stage of its application modernization journey. IBM Client Engineering is an investment in you to co-create and innovate by leveraging IBM technology and methodologies.

What you can expect

- Solve complex business use cases: Develop user-centric solutions to your most pressing challenges with measurable business outcomes.
- Prove value in weeks: Together, we define the proposed solution and deliver value in days to weeks.
- Innovate to scale: We co-create iteratively to grow alongside your organization, delivering enterprise scalability that is securely deployed on your platforms of choice.
- Partner with global experts: Leverage deep industry expertise with our agile teams of business technology leaders, technology engineers, solution architects, designers, and data scientists.
- Modernize with speed: Build rapidly with IBM hybrid cloud and AI technologies to stay ahead of an ever-changing market.
- Access proven patterns: Advance digital transformation with IBM proven technology, accelerators, and methodologies.

Conclusion

To accelerate your digital transformation, you must increase productivity and business agility, and close skills gaps. To achieve these tasks with z/TPF and cloud, IBM recommends the following best practices:

- Embracing a *hybrid cloud* approach to z/TPF application modernization
- Leveraging a continuous application modernization journey
- Leverage the z/TPF reference architectures to achieve your modernization goals

Use this IBM Redbooks publication to review the different entry points and select the ones that drive the most immediate value for your business depending on your needs, environment, challenges, and current business processes.

By partnering with IBM and following our recommended application modernization journey, we can help you modernize z/TPF applications faster, at lower cost and with less risk. The approaches and strategies that are presented in this publication can help you to accelerate your digital transformation and increase productivity and business agility.

Authors

This publication was produced by a team of specialists from around the world working with IBM Redbooks, Poughkeepsie Center.

Mark Gambino
IBM® Distinguished Engineer - z/TPF Chief Architect

Luanne Jagich
IBM, Program Director, z/TPF

Jamie Farmer
IBM, Senior Software Engineer - z/TPF

Kelly Paesano
IBM, Manager, z/TPF

Connie Walberg
IBM, Product Manager - z/TPF Family of Products

Josh Wisniewski
IBM, Senior Software Engineer - z/TPF

Thanks to the following people for their contributions to this project:

Lydia Parziale, Certified IT Specialist, PMI Certified Project Manager, IBM Redbooks Project Leader
IBM Redbooks, Poughkeepsie Center

Wade Wallace, Senior Content Specialist and Editing Team Lead, IBM Redbooks
IBM Austin

Notices

Trademarks

REDP-5714-00

ISBN 073846130x