DGP's hub-and-spoke architecture is deployed as a two-tier thin client/fat server topology by grouping the logic hub and the data storage spokes together within the server tier. When it comes time to build an implementation of the DGP architecture, it can be broken down into 4 relatively independent logical tiers:

1. The thin client applications and remote integrated system spokes are completely independent of the APIs thanks to the generic API messages, and can be built using any programming language that is able to create API request messages, POST them to the server, and read the API response messages that are returned. SOAP web services are another option that are used mainly by the optional automated processing subsystem for their one-way request/acknowledge protocol.
2. The structured text (XML fragment) API messages provide generic compatibility between many different types of client applications and integrated systems when calling the server APIs, with no serialization/deserialization and no need for client SDK's to be maintained for each supported programming language. Good documentation of the APIs replaces the client SDK's.
3. The web server tier hosting the message-based web service APIs. This is the hub for almost all of the logic in a system.
4. The data storage tier spokes which can use standard database schemas and clusters, or DGP replication using special schemas across multiple locations, or a mix of both.

All of the different tiers and parts of the modular multi-tier system use various types of RPCs for inter-process communication, each of which are based on configurable endpoints. This is the basis for the flexible and relatively independent tiers that are easy to evolve.

At a high level, DGP's modular multi-tier hub-and-spoke architecture behaves like a two-tier thin client/fat server system. The code of the client applications (and remote integrated systems) is totally independent of the server tier APIs in terms of development and deployment thanks to the generic structured text API messages combined with the immutable append-only convention, and can therefore be maintained on their own schedules.

The server tier has its own logical and physical tiers, and several levels of modularity (the spokes in the hub-and-spoke architecture, the APIs and code libraries inside the logic hub implemented as Visual Studio projects, etc.). System evolution capabilities based on the immutable append-only convention allow for the constant addition and modification of a system's functionality without ever breaking backward compatibility – to the point that the easiest way to build a new DGP system is to append new APIs and databases (storage spokes) to extend an existing DGP system, and then add new application spokes to call the new APIs.

The purpose of this documentation is to highlight the various implementation options that are available, and to explain the main reasons why a particular combination of tools was selected for the DGPDrive reference implementation.

DGP NFR Summary

DGP has implemented functionality that meets a set of Non-Functional Requirements (NFR's) that far exceed the needs of most business software systems. DGP's NFR's are described in more detail within their respective sections of the requirements documentation. When evaluating alternative architectures and designs, it is important to test (prove) their ability to meet the sum total of all of the product/service requirements plus all of the NFR's for a given software system. The summary of NFR's listed below also contains brief descriptions of the atypical DGP/DGPDrive design and implementation techniques that have been proven to best meet all the requirements through champion/challenger competitions.

- Strong, effective security – refer to the security section of the documentation for more details
    - Strong network perimeter security
    - An identity store that provides data-driven Role Based Access Control (RBAC) authorization functionality
    - Simple, fast and reliable mechanisms for both account authentication and authorization to control access to system resources, functionality and data (based on the security data managed by the RBAC identity store)
    - Security controls for message TTL, account rate limits, failed authentication counts and lockouts, password resets, etc.
    - A scalable, transparent and easy to maintain mechanism to manage the encryption of system secret data (connection strings, system account credentials, etc.)

    - ✓ *Message-based RPC-style APIs using a single-argument front controller pattern reduce the exposed attack surface of a system down to a single method with a single input parameter.*
    - ✓ *Messages allow a great deal of information to be sent back and forth between the client and the server. This is used to embed security information into each API message.*
    - ✓ *All of the functionality of an API Security Gateway are consolidated into the single front controller method that all API messages must pass through. This reusable method implements all checks, counts and tests of the input values, etc.*
    - ✓ *The security of the messages themselves are improved by using XML fragments and avoiding serialization/deserialization.*

- ✓ *A simplified custom identity store database, optimized for very high performance, is used for both message authentication and the ACL's (Access Control Lists) needed for API method authorization.*
- ✓ *Encryption of sections of .NET configuration files for management of secret system info (connection strings, credentials, etc.)*

- High performance (.NET) – all API calls must return a result to the calling application in less than one second, end-to-end.

  - ✓ *Minimize the rate of object instantiation to reduce the workload of the GC.*
  - ✓ *Create the smallest, flattest graph of reachable objects (object dependencies) to simplify the workload of the GC.*
  - ✓ *Restrict the size of objects so that they are created in the small object heap (less than 85,000 bytes) and not the LOH.*
  - ✓ *Ensure objects are short-lived and collected in Gen0 or Gen1 memory segments, and are not promoted to Gen2 segments.*
    - *Break up large, long-running processing tasks into small batches run iteratively, which helps reduce the size of the objects and insures that the short duration of the small batch enables Gen0 or Gen1 collections.*
  - ✓ *Avoid "expensive" .NET syntax and instead use the lowest level, simplest elements of the .NET Framework.*
  - ✓ *Read API messages using a forward-scrolling "firehose" cursor through a memory stream (avoid serialization/deserialization).*
  - ✓ *Cache data within the database server (identity store records) and web services (ASP.NET Cache object).*
  - ✓ *Use the ASP.NET Cache object to cache values, and set the appropriate expiration interval.*

  IMPORTANT NOTE: End-to-end performance is one of the most important and useful measurements of a distributed system. Achieving high levels of performance ***forces*** *the simplification of the architecture and code. It is impossible for over-engineered, overly complex code (and designs) to deliver high performance results. Therefore, measuring and tracking the performance of each API method over time (using the API Tester test harness) is one of the best and easiest ways to insure that low-quality code is not allowed to be added to a DGP system as it evolves over time.*

- High efficiency (.NET) – the web service APIs must maintain high levels of performance and low levels of resource utilization over long periods of continuous use; in other words, no memory leak behavior, no excessive use of CPU, network or data storage, and no slowdown in performance over time.

- ✓ *This requirement includes most of the same objectives listed for the high-performance requirement, but the performance and resource utilization of the web service APIs must be measured over long duration load tests. The efficiency of the web service APIs has a large impact on the scalability (and overall usability) of a DGP system.*

- High availability – 100% uptime for the system with sufficient redundancy – individual servers or entire locations can go offline (planned or unplanned) while the system as a whole continues to function uninterrupted.

  - ✓ *Redundancy with automatic isolation of failed nodes, failover of work, and automatic recovery of failed nodes after they have been repaired (automatic recovery of data replication is particularly important).*
    - *Processing: stateless web server farms provide fault-tolerance for the web service APIs and web applications.*
    - *Data storage: DGP omnidirectional data replication between locations can provide fault-tolerance and 100% uptime for replica databases.*
    - *Data storage: SQL Server Availability Groups can provide fault tolerance within locations for all databases (compatible with DGP replication).*
    - *Data storage: SQL Server Availability Groups configured to span across multiple locations can provide fault tolerance and 100% uptime for all databases.*
  - ✓ *Partition tolerance so the system can continue to function during periods with offline segments.*
  - ✓ *Partition tolerance enabling periodic maintenance of segments of a system with no system downtime.*

- High scalability – the system must be able to scale both processing power and data storage up to maximum levels that are well above the needs of most organizations, which allows systems to start small, grow as needed, and never outgrow the system's original architecture.
- Processing node web service API methods should have an average server-side processing time of less than 10 MS.
- Processing node web servers should be able to handle a minimum of 100 requests per second per CPU core.

  - ✓ *Processing: stateless web server farms provide incremental horizontal scalability for the web service APIs*
  - ✓ *Data storage: DGP database shards provide incremental horizontal scalability for some (not all) replica databases.*

- ✓ *Data storage: SQL Server vertical scalability (more CPU cores, more memory, more solid state storage) can be used for all databases.*
- ✓ *Use a lazy, JIT mechanism to update the ACL data cached in each identity store account record.*
- ✓ *Windows IO completion ports manage the thread pools for all IO-bound RPC's to web services, database servers, etc.*
- ✓ *The long duration performance and efficiency of the web service APIs have a big effect on the scalability of software systems.*

- The ability to run all of the tiers of an entire system on a single computer is required for software development, testing, debugging, and to run small-scale production environments.

  - ✓ *DGP uses a hub-and-spoke logical architecture merged with a basic multi-tiered distributed system architecture in order to meet both the requirement for all the tiers of a system to run on a single computer, and at the same time be able to scale up to levels well beyond the needs of most organizations (refer to the architecture comparison below for more details).*
  - ✓ *The ability to run all of the tiers of a distributed system on a single computer not only simplify and increase the productivity of software development, but it also allows any problems with the system to be debugged and identified within minutes (.NET).*

- Automated testing – testing built into the architecture of the web service APIs that allows both ad hoc and automated full regression tests (a necessary prerequisite for frequent/continuous deployments).
- Long duration load testing – the measurement of the end-to-end performance and resource utilization (memory, storage, network) over long periods of high volume workloads.

  - ✓ *Unit tests and server-side performance measurement are permanently built into every API method. The results of the unit test and the server duration time are returned as part of every API response message.*
  - ✓ *Client applications display the results returned by the API methods, and the end-to-end performance of the API call, and the server-side performance of the API method; if an error occurs, the error message from the server-side unit test is displayed.*
  - ✓ *Members of the RemoteMonitor role save the result codes and performance measurements to the location they are using.*
  - ✓ *The API Tester test harness can run full regression tests of every method in an entire system remotely, just like client applications; the results of a test run can be saved to the location of the environment being tested for analysis over time.*

- ✓ *The API Tester can continuously repeat a set of tests for a specified number of iterations as part of load testing; ongoing results of the repeated tests can be saved for analysis.*
- ✓ *Various tools can be used to monitor and record both the performance and resource utilization of the web service APIs, web servers and database servers during load tests.*

- Automated processing subsystem – a scalable subsystem to run iterative processes continuously in the background, with full security, monitoring and recovery functionality; all automated processes are implemented as API methods; (optional subsystem).

  - ✓ *The automated processing system is used by default to run the automated data replication between locations. If standard data storage is used, this subsystem is completely optional.*
  - ✓ *It is an extensible, data-driven subsystem. All automated processes are implemented as autonomous SOAP one-way API methods. One or more generic scheduler Windows services read the queue of scheduled processes to execute the iterations.*
  - ✓ *Complex processes with state information are maintained in their own schedule tables. The queue records maintain process state from one iteration to the next, and also act as an authorization token to guarantee sequential iterations.*
  - ✓ *New automated processes can be added to the web Service APIs, configured in the schedule queues, and run by the scheduler services as needed, with no modifications to any existing code, schedulers, etc.*
  - ✓ *The automated processing subsystem is very scalable, performant, fault tolerant, and observable. It automatically fails over and recovers from most problems, and documents each process iteration to various logs.*
  - ✓ *The scheduler has adaptable iteration scheduling to work through backlogs at maximum speed, and resume normal iteration intervals once it has caught up.*

- Simple deployments and rollbacks – the software must be easy to deploy and just as easy to roll back whenever necessary.

  - ✓ *Deployments are based on the .NET XCOPY methodology. Applications are deployed as folders below a parent directory.*
  - ✓ *Folders require no installation. Apps are uninstalled by deleting the appropriate folder from storage.*
  - ✓ *No folders are ever merged when deployed, following the immutable append-only convention.*
  - ✓ *The application path to the executable (.NET assembly) is modified to use a new release folder.*

- ✓ *Rollbacks only require the executable path to be reset back to the previous release folder, which remained unchanged.*

- System and service evolution that always preserves backward compatibility – the constant extension, enhancement and fixes made to a system (evolution) must never cause breaking changes to the applications and integrated systems that depend on the API functionality.
- Zero maintenance of API methods, associated test files and documentation pages once they have been put into production (thanks to the immutable append only conventions).

- ✓ *All functionality in a DGP system is built as web service API methods. Those API methods follow an immutable append-only convention that is guaranteed not to break backward compatibility in spite of constant changes to the system.*
- ✓ *Once an API method is deployed to a production environment and used by client applications, integrated systems, etc. it becomes immutable. The test file for that API method (used by the API Tester test harness) and the documentation page for that API method are also effectively immutable since the API method never changes.*

- System data management subsystem
  - Configuration data
  - Secret data (system account credentials, database connection strings, and other sensitive data) – this functionality is the equivalent of an HSM to protect sensitive configuration data in the system

- ✓ *App.config and web.config files are used to manage and maintain configuration data (ASP.NET).*
- ✓ *The full version of the .NET Framework provides the capability to encrypt sections of the web.config file to protect sensitive data for a system.*
- ✓ *The configuration and secret data can be easily and securely shared between multiple separate locations, since they are just XML files.*

- Data caching – caching built into the DGP web service APIs in addition to the caching in the database servers and web servers.

- ✓ *Static data or data that changes infrequently is cached using the Application object (ASP.NET).*
- ✓ *Dynamic data and data that expires uses the Cache object (ASP.NET).*

Tenants, Security, Redundancy and Replication

Each "tenant" of a DGP system is the **owner** of a set of infrastructure used to host the various tiers of their DGP system, the web services hosted on the web servers, the data stored in its various database storage spokes, and at least some of the client applications that use the functionality of the system's API methods (the hub). At a small scale, all the tiers of a DGP system can be hosted on a single computer. At larger scales, each tier and more and more individual parts of each tier are partitioned and hosted on their own separate hardware resources. The infrastructure itself consists of web servers hosting the web service APIs, and database servers hosting the collection of DGPDrive databases (using the DGPDrive reference system as an example).

Each type of infrastructure has its own levels of security – for example the security of the host OS for each server, the security of the database engine, and so on. The host servers can be managed as part of an Active Directory domain, or as individual servers. In a DGP system, the only part of this lower-level security that matters are the database engine (SQL Server) accounts that are used by ADO.NET connection strings to provide system-level access for the web service APIs to each DGP database.

Within this context, DGP-level security (the data-driven RBAC subsystem used for authentication and authorization) is provided by the SysInfo database, which is a custom identity store consisting of user accounts, API methods, roles, role membership, data security groups, etc. This custom identity store is not meant to be partitioned to support multiple "sub-tenants".

Looking at a DGP system from this perspective, each environment of a system can be thought of as its own separate "tenant" since it has its own separate web server, web services, database server, collection of DGP databases and custom identity store. In this sense, each developer workstation is basically a single-server installation that hosts its own version of all the tiers and databases of a DGP system, either a stand-alone system or as a location of a larger distributed system. The only real difference is whether or not the data in a location is being replicated and synchronized between multiple locations (in particular the security data in the SysInfo database).

When joining together multiple locations as parts of a single distributed grid system (vs. running as a standalone location) the only difference is the synchronization of the data between the multiple locations – especially the security data in the SysInfo database.

This can be accomplished by using SQL Server Availability Groups or with DGP omnidirectional replication. DGP's DBSetup application is used to both build and maintain the database schemas for the multiple locations and environments of a system, and also to programmatically "replicate" core security data that must be synchronized between locations for a distributed grid configuration to function properly. The core security data is added to a location by the DBSetup application as if it had been replicated from a master database (which in reality only exists logically in the code of the application). This allows a consistent set of core security data to be inserted into new locations in a way that is compatible with the data replication between locations, which then allows an admin account to log into a new system and begin to create other accounts in the identity store, etc. (effectively a bootstrap mechanism for new locations). This can also be used to synchronize data between locations and environments.

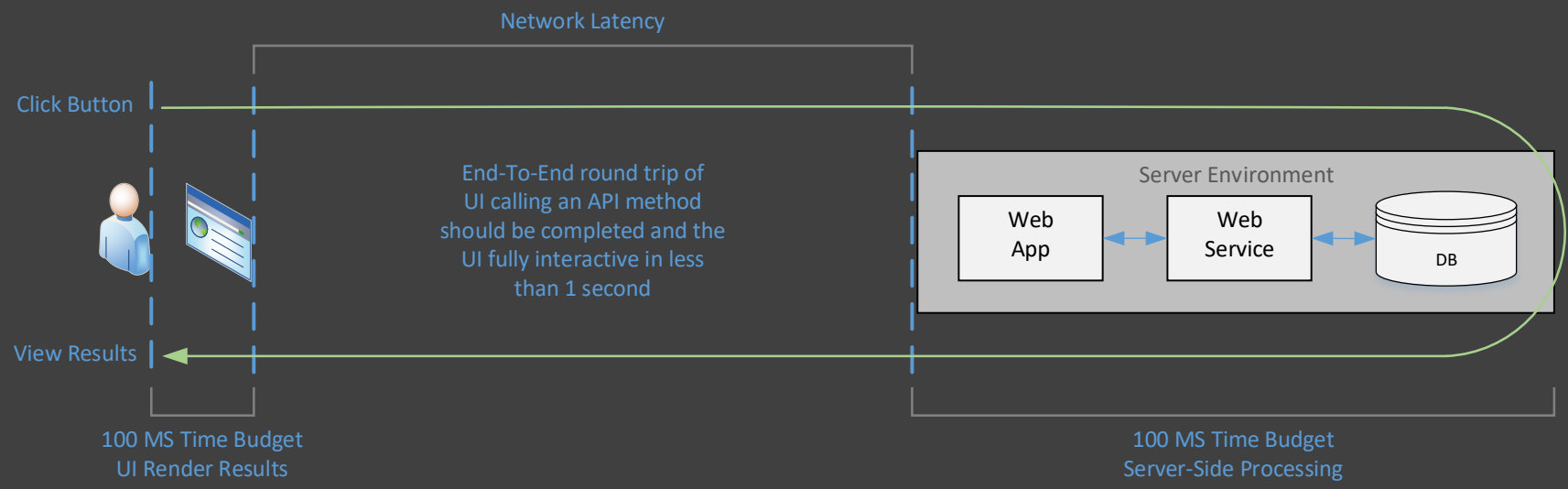
By default, a DGP/DGPDrive system is not meant to be a shared, centralized infrastructure with multiple logical tenants sharing the resources of a single DGP system. It is meant to be distributed and decentralized, with each individual tenant owning and controlling its own set of hardware resources, generally duplicated in more than one location in order to deliver 100% system uptime.

Functional Prototype Competition

The choices of operating systems, programming languages, development tools, database engines, etc. used for DGPDrive were made based on data collected from tests and measurements of prototypes built as experiments for the major alternatives.

NFR's (non-functional requirements) and product requirements are documented in advance, and are used as a guide to design and build the various prototypes. When completed, the prototypes are tested and measured to see how well they meet the total set of all of the various types of requirements. The results are then compared head-to-head among the different prototypes in order to determine which alternative delivers the best test results, at the lowest cost, and in the shortest amount of time.

In order to meet DGP's requirements, the implementation must deliver API functionality as natively compiled binary executables which are very fast and efficient (interpreted languages are not competitive in terms of performance and security). All logic in a DGP system is consolidated in the logical hub on the server tiers, implemented as message-based web service APIs.



One of the best measurements of system performance is the end-to-end duration of client applications calling the web service API methods. This performance metric can be used to fairly compare the performance of different types of software systems. The performance standard for DGP systems is that every API call should return results to the client app in less than one second. Within that 1 second end-to-end time budget, the maximum duration allowed for this server-side processing is 100 MS.

The server-side processing of DGPDrive web services (Windows .NET Framework 4.8.1) currently averages 10 MS with no web service API-level caching, and approximately 2 MS with API caching enabled, once the web service cache has been populated. This level of performance helps to improve scalability, increasing the number of API request messages that can be processed sequentially by each web server core, per second.

An additional objective related to performance is efficiency. The standard for DGPDrive web service efficiency is that the server performance starts out high and remains at that same high level over time no matter how long the web services are run. Similarly, resource utilization on the server will gradually increase to a steady state based on the volume of the server workload, and will not increase any further over time no matter how long the web services are run (in particular, no “memory leak” behavior).

Many systems do not behave this way. It is quite common for the performance of web services to degrade over time, while the resource utilization (especially CPU and memory) continuously increases over time – behaving as if the system has a memory leak. DGPDrive is designed to deliver high performance, good efficiency and good memory management in order to avoid these types of poor results. To do so requires architectures and techniques that are different from the current status quo "best practices".

There are many requirements for DGP's server stack at the system and product/service levels (refer to the requirements documentation for more details). In terms of software development, the relevant requirements for the choice of server stack programming languages are:

1. Good developer productivity
2. Able to produce natively compiled executables with extremely good performance and efficiency
3. Strong, effective, multi-layered security
4. A mature, mainstream language with a large pool of developers
5. (optional) Cross-platform compatibility – which is essential for client applications, but not is important for the server stack

Within that context, there are 3 main choices for the programming languages used for the server stack itself:

- Interpreted languages such as JavaScript, Typescript, Python, etc. These languages tend to have the best developer productivity, but their performance and security are both lacking. Interpreted languages cannot match the performance of natively compiled binaries, and their security is substandard because the source code itself is deployed to be interpreted (and the source code can be edited on the server, which is a serious security problem).
- Managed-code languages such as Java, C#/.NET, and Go. These languages have been proven to offer the best combination of developer productivity combined with good performance and efficiency. In addition, the natively compiled binaries (when applicable) are immutable and cannot be edited on the server, which helps improve security.
- Low-level languages such as C/C++, Rust, etc. While offering extremely good performance and strong security, the developer productivity of the low-level languages is lacking compared to the other options.

From this perspective, the managed code languages offer the best balance of high performance, good security and good developer productivity.

Choice of Operating Systems

At a high level, DGP's modular multi-tier architecture behaves as a loosely coupled 2-tier thin client/fat server architecture. In other words, the generic API messages allow the client application to be developed independently from the server tier, so client applications can be built using just about any combination of programming language and operating system desired – without the need for any client SDK's.

On the server, DGP systems have the same modularity as micro services, but the modular functionality is built as reusable libraries of code that can be shared by many web service APIs, and are able to call each other within the same memory space on each web server whenever possible. This greatly increases the performance of inter-process communication, simplifies server-side security, and verifies all dependencies between libraries at design time when the services are compiled – instead of at runtime after micro services are deployed.

The choice of operating system for the server tier is more significant, and really comes down to 2 options: Windows or Linux. Purely from an operating system perspective, they are very similar in terms of total cost. Windows costs more for software licenses, support costs are about the same, and the cost of Windows technical staff is lower. Linux saves on the cost of software licenses, but the technical staff typically cost more – so the total cost is about the same. The cost of the technical staff is by far the largest single component of the total cost of a software system, dwarfing the costs of hardware, infrastructure, software licenses, etc.

To allow for flexibility, DGP's architecture is designed to be relatively agnostic in terms of the operating system, programming languages, web servers and database engines that can be used to build the server tiers.

In reality, the choice of operating systems, web servers and database engines depend mainly on the knowledge and skillset of any existing technical staff within each organization.

Windows Server	Linux Server
Summary: from the perspective of total cost, paying more for software licenses while paying less for technical staff – all other things being largely equal.	Summary: from the perspective of total cost, paying less for software licenses while paying more for technical staff – all other things being largely equal.

Advantages: • The best ease of use, especially for administrators • Huge installed base of users familiar with and trained in the use of Windows (for many years) • Largest market size of software that runs on Windows • The best automatic updates and patching of the major operating systems • The full .NET Framework is built into Windows, and is automatically supported/patched as part of Windows • The best overall security (and security tools market) of the major operating systems • I/O completion ports for improved efficiency of thread pool thread management (important for web service APIs) • The ability of the full .NET framework to encrypt configuration files using Microsoft's PKI (important for web service APIs)	Advantages: • Low cost of software licenses and to a lesser degree, support contracts • No artificial limitations built into the operating system • Good performance and efficiency • Many choices among different Linux distros, shells, GUIs, etc. • Open-source code, which can provide many benefits • Many open-source applications are designed and built primarily for Linux
Disadvantages: • The high cost of software licenses • The artificial limitations built into the various editions • The need to put up with the many stupid things Microsoft does periodically (too numerous to mention) …	Disadvantages: • Lower ease of use, especially for administrators • Higher overall level of knowledge, experience and competence required by administrators (greater workload) • Smaller market for software that runs on Linux • Weaker overall security compared to Windows • Weaker patching and update process compared to Windows • Smaller installed base of users familiar with and trained in the use of Linux

Web Service Options

DGP can use both HTTP and SOAP web services, but does not support REST web services. The single argument front controller pattern with its single web service endpoint and single front controller method is very important for the message-based APIs, security,

performance, etc. and is basically the opposite of the REST style of building web services that have discrete URL endpoints per individual resource. Also, the single argument front controller eliminates one of the most serious weaknesses of SOAP web services, which is the fragile nature of the WSDL, stubs and proxies and their inability to gracefully handle the evolution of APIs and methods.

DGP's message-based APIs only need a single front controller API method whose signature literally never changes (all variability and evolution is encapsulated within the API messages themselves). The SOAP protocol is the only option that provides the request/acknowledge pattern of one-way methods, which is needed by the DGP automated processing API methods. HTTP web services simply POST the API request message as the HTTP body – which works well, but HTTP does not support the one-way request/acknowledge pattern. This one-way controller is essential to allow the web server to handle multithreading for the automated process API methods at scale. The prototype which built the multithreading into the AutoWork Windows services did not deliver good results at scale compared to the multithreading capabilities of the web server using SOAP one-way API methods.

DGP Client Application Options

The development of client applications is independent of the server APIs thanks to the generic API messages, and therefore have a great deal of flexibility in terms of the combinations of OS, platforms and programming languages that can be used to build them.

The only requirements for the client applications are:

1. The client applications are able to create XML API request messages and read XML API response messages (no client SDK required).
2. The client applications are able to use an HTTP POST to send the API request messages to the DGP web services, and/or are able to call a SOAP API method.
3. The client applications manage their own user state data.
4. (optional) The client applications are able to use DGP's built-in API message encryption, which is a substitute for TLS.

Cross-platform native applications are generally preferred instead of web applications for their better performance and security (and in some cases, additional functionality). React Native, Maui, Uno, Flutter, .NET Core WinForms, and a number of other alternatives are some of the good options available for building the thin client applications.

Choice of API Message Format

The messages in DGP's message-based APIs are very important to the functionality of the overall system. Among other things, they provide a 2-way communication mechanism to share a great deal of information between the client apps and the server APIs. Many of the innovations provided by DGP depend upon these capabilities. The requirements for API messages are:

1. Strong security
2. Extremely high performance
3. Able to transport any type of data without the need to escape or encode special characters
4. Generic compatibility across all major programming languages and platforms
5. No client SDK required for client applications to be able to use the API messages

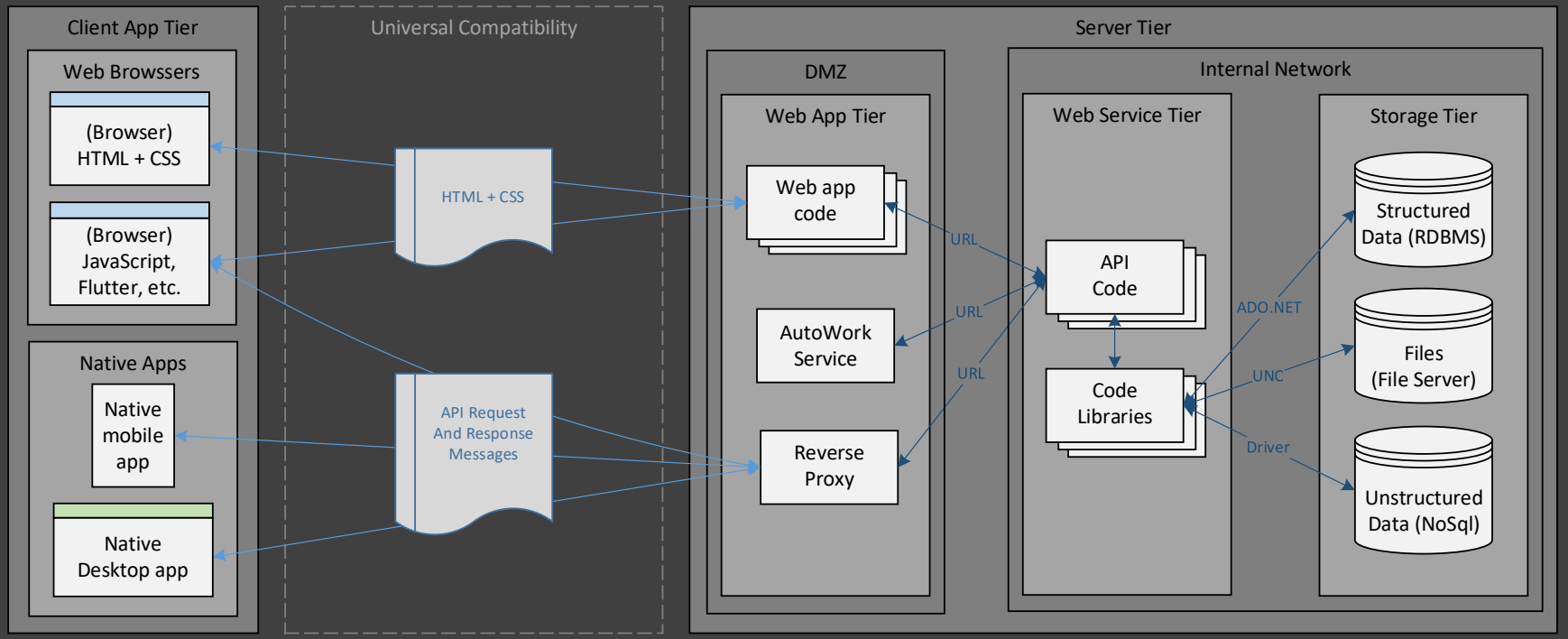
The two main choices for the message format include XML and JSON (Protobuf is another potential alternative, but it is not natively supported in most platforms). XML prototypes were found to be superior to JSON for the following reasons:

- CDATA blocks allow any type of data to be contained in the XML messages without encoding or escaping special characters.
- The event-driven memory stream reader is approximately 140 *times* faster than the fastest serialization/deserialization.
- The event-driven memory stream reader is the foundation for the tolerant reader functionality in DGP.
- XML can only contain data (no code injection possible) which helps to improve security.
- XML fragments eliminate strong data types, external entities and other XML vulnerabilities to help improve security.
- Support for XML is already built into most of the major programming languages.

It is important to understand that DGP's API request and response messages are NOT treated as serialized objects, and this fact helps to improve both performance and security, while also significantly reducing the amount of code that needs to be built and maintained for each system. If custom types were used, each custom type has a static structure, so each API method would require a pair of custom types to be built (one for the request and one for the response), maintained, versioned, etc. – generally as part of a client SDK created for each supported programming language.

Instead, DGP's XML API messages are handled as flexible collections of message elements. Request messages contain a collection of one or more API method calls, each of which contains a collection of zero or more name/value pairs representing the method input

parameters. API response messages contain a collection of one or more method results, each of which can themselves contain a wide variety of different data. All input parameter values and method result values are encapsulated within CDATA blocks.

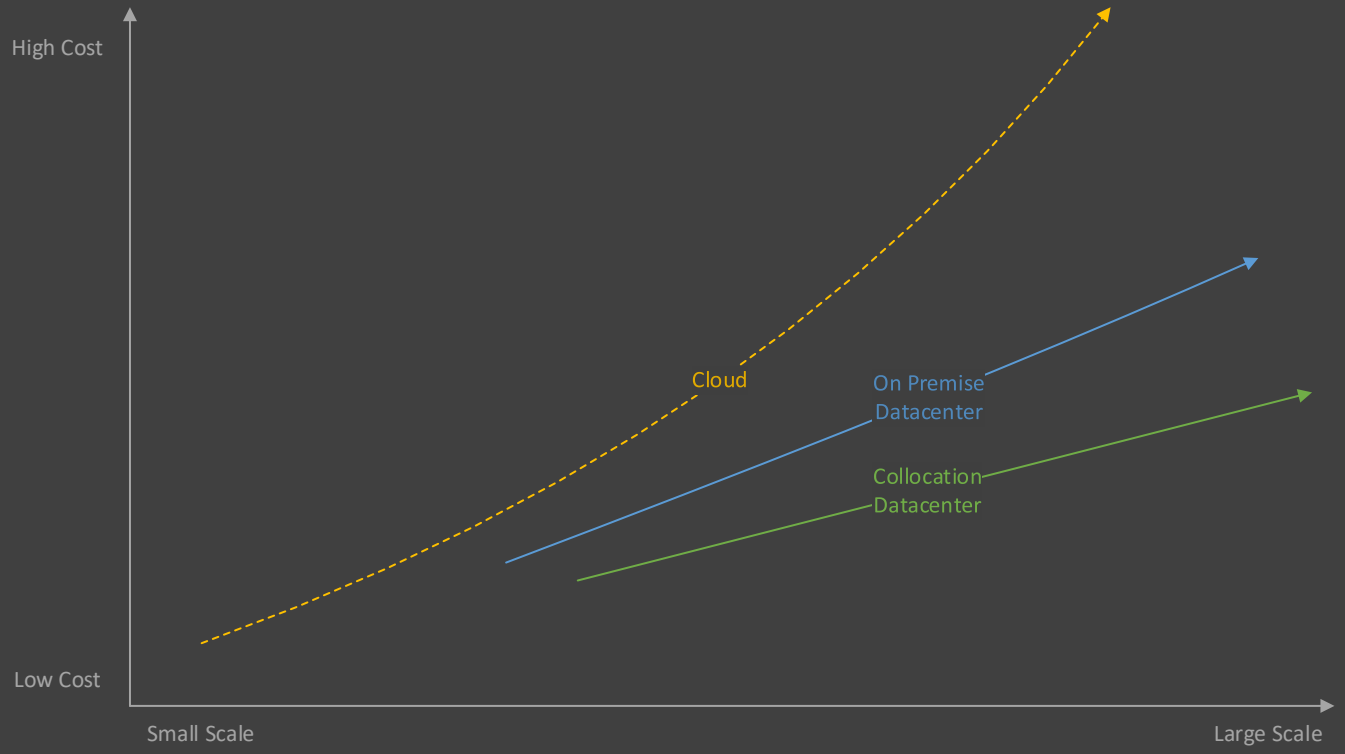


Using a single flexible, generic mechanism capable of handling any structure of request and response messages, rather than building custom types for each specific API method request and response provides multiple benefits:

- The elimination of the need to build and maintain hundreds or thousands of custom request/response message types significantly reduces the amount of development work for a system, saving a great deal of time and expense.
- A very simple template mechanism is used to create the XML fragments for the API request and response messages in a standard, consistent way by a reusable utility library.

- The XML fragments are read using a single pass of a forward scrolling cursor through a memory stream. There is no faster way to read messages, and this process is approximately 140 times faster than the current fastest serialization/deserialization.
- The lack of strong data types enforced in the messages helps to improve security by eliminating any potential exploits in a strongly typed serialization/deserialization process. Instead, the input values are verified and cast to the correct data type at runtime as they are read.

Choice of Infrastructure



The objective of DGP implementations is twofold. First, to allow all parts (tiers) of a system to be installed on a single computer, and then to easily scale both vertically and horizontally with no changes to the system other than the configuration of the RPC endpoints. Second, to allow cloud, on premise and collocation implementations to all be identical in terms of setup, configuration, deployment and maintenance, thereby allowing the maximum degree of flexibility in terms of the infrastructure used, while also avoiding any type of vendor lock-in that would prevent taking advantage of that flexibility.

Cloud	On-premise	Collocation
Summary: Running your software on somebody else's shared hardware, in a datacenter you will never see.	Summary: Running your software on your own hardware, in your own datacenter.	Summary: Running your software on your own hardware, in a rack you rent in somebody else's datacenter.
Advantages: The main advantage of cloud infrastructure is elasticity, namely the ability to easily add more resources as needed, and remove resources when they are no longer needed. Some economies of scope are also possible, depending on the vendor.	Advantages: the best performance and scalability of the resources plus the benefits of owning those resources (depreciation). Also, the overall security of the On-premise infrastructure can be very good.	Advantages: all of the same advantages of On-premise provided at the lowest total cost. In addition, economies of scope and scale can result in a higher level of quality and reliability for the capabilities of the datacenter itself.
Disadvantages: The highest cost of infrastructure, combined with the lowest performance and scalability of each resource due to the fact that all resources are virtual, sharing the underlying physical resources with other customers. The security of customer data and intellectual property can also be an issue.	Disadvantages: the primary disadvantage is the lack of elasticity (the need to plan ahead to buy additional hardware in advance of when it is needed, and the inability to simply turn off infrastructure when it is not needed). Another disadvantage is the second-highest cost of the infrastructure, largely due to the high cost of duplicating all the functionality of a collocation datacenter yourself On-premise.	Disadvantages: the same elasticity disadvantages as On-premise, with the possible addition of travel time to the collocation datacenters periodically.

DGPDrive Beta Reference Implementation

The DGPDrive beta prototype has been built using Visual Studio and C#/.NET for both the client app and the web services. The Windows prototype uses the full .NET Framework 4.8.1, which is focused on both stability and backward compatibility. It has a few useful features that .NET Core does not have such as encrypted configuration files, which can act as a replacement for HSM's when needed. It will not be adding any new features going forward, but some performance improvements from .NET Core will migrate to the .NET Framework when applicable. The .NET Framework is automatically updated by Windows Update, which is very important for the security of the system. Finally, the .NET Framework 4.8.1 is considered to be a part of Windows itself, and is guaranteed to be supported for every version of Windows going forward – with no "end of support" dates like there are for every version of .NET Core.

Given the fact that DGP is targeted at the US SMB market, and the Windows OS has over 90% share of that market makes the .NET Framework 4.8.1 is a good choice to build all of the tiers of the DGPDrive beta/prototype. It allows for rapid application development, which is important for a system designed, built, tested and documented by a single individual. It contains all of the needed functionality, and requires no 3rd party tools or libraries, which is good for security. In addition, the option for encrypted configuration files is a very useful security feature that is not available using any other platform. Finally, the stability and reliability of the .NET Framework 4.8.1 is a good match for DGP itself which emphasizes reliability, stability and high quality. Prototype web service controllers have been built using .NET Core 5 with no problems, so if cross platform compatibility on the server becomes a requirement in the future, the migration to .NET Core would not be an issue. However, Java is probably an even better choice for cross-platform server tier development than .NET Core. Prototypes would be used to prove which one delivers the best results.

Whenever Windows and the .NET Framework are viable options, then the fastest way to build a "new" DGP system is to begin with the code base of an existing DGP system (such as DGPDrive), and then to append new APIs to the hub, along with new database and client application spokes in order to implement all of the new functionality. DGP system evolution is designed to work this way, allowing each DGP system to act as a growing collection of APIs offering a wide range of functionality (which imitates the namespaces of the .NET Framework itself). The data-driven RBAC security system provides fine-grained control over exactly which methods each account is authorized to call based on role membership. This in turn allows each client application to be built to only use a subset of the growing collection of APIs, in much the same way that .NET applications only use a small subset of the namespaces available in the .NET Framework.

.NET Memory Management

Memory management is crucial to the high performance and efficiency of the compiled native binary executables. Memory management when using a managed-code tool (which automates memory allocation and deallocation) consists of designing and building your code in a way that minimizes the amount of work it creates for the garbage collector (GC) to clean up afterwards. For the .NET Framework 4.8.1, this can be achieved in the following ways:

1. Design objects to be as small and short-lived as possible, so that they are created in the small object heap and collected in Gen0 or Gen1 memory segments.
2. Avoiding the use of the LOH by insuring objects do not exceed 83K in size. This is accomplished by breaking up large processes (and their associated data) into incremental iterations. Server-side data pagination is one example, limiting the maximum number of records returned within a page of data. Automated processes that continuously operate incrementally on small batches of data is another example.
3. When small short-lived objects are not possible:
    a. Reuse long-lived objects in Gen2 segments instead of instantiating new ones.
    b. Reuse large objects in LOH segments instead of instantiating new ones.
4. Minimize the rate of object instantiation as much as possible. There is a cost to object instantiation and collection, so the fewer objects in the managed heap, the less work there is for the GC. The methodology used to reduce the rate of object instantiation is another form of consolidation. Instead of strictly following the mandates of object-oriented design to scatter code widely among a large number of classes and methods, code is instead consolidated into far fewer methods. The general guideline to follow is that each method should contain all the code needed to perform an action or task to completion.
5. Create the smallest, flattest tree of object dependencies possible. Large trees of dependencies many levels deep are difficult for the GC to trace from their roots and often result in references to objects that are held longer than necessary. Avoiding inheritance is another key technique to flatten the treed of object dependencies. In general, classes should be treated as independent self-contained libraries of code that mimic the namespaces in the .NET Framework. The objective is to avoid the "all I wanted was a banana, but I got a gorilla holding a banana along with the whole jungle" dependency problem.

From a code perspective, these practices used to minimize the work of the GC result in a hybrid of OO techniques used to organize the code into classes and methods, combined with procedural programming for the larger amounts of code consolidated within the

smaller number of methods. The other parts of OO such as inheritance, polymorphism, etc. are avoided in order to achieve better performance and efficiency.

These hybrid practices are basically imitating some of the techniques that were used to improve the simplicity, performance and efficiency of Google's Go language, which is also a non-OO hybrid. The benefits of these hybrid techniques when used in C# is that only a few changes are needed in order to achieve significantly better performance and efficiency, while still using .NET instead of Go.

However, the overall philosophy for DGP systems is that whichever combination of programming language and techniques deliver the best results that are able to meet all of the requirements, at the lowest cost and in the shortest amount of time becomes the new champion to be used going forward (until a different technique becomes the new champion, and so on). This choice can be affected by outside factors such as developer availability, etc.

Windows 10 Limitations

Windows 10/11 is suitable for software development, testing/debugging/troubleshooting, and for small-scale systems. The primary limitation is IIS on Windows 10/11 only supporting 20 concurrent HTTP connections. For small DGPDrive systems, this mainly affects the web service APIs. However, the APIs are stateless, with very high performance (short duration), so the 20 concurrent connections can support a reasonable workload for a relatively small number of users. The upgrade for Windows systems would be to Windows Server Standard edition. A good alternative would be for the web services to be built in Java, hosted on Linux.

SQL Server Express Limitations

SQL Server Express is limited to 4 cores, 1 GB of RAM, and 10 GB database size. Once again, these limitations are suitable for small-scale systems, especially when using the database shard option for the content stored in DGPDrive. The upgrade for Windows systems would be to SQL Server Standard edition. A good alternative would be MySQL or Postgresql, hosted either on Windows or Linux.