What is DGP (Distributed Grid Platform)?

DGP is a free, open-source, open-documentation software development platform/framework that is intended to help build new custom software systems quickly and inexpensively by reusing all of the features, functionality and capabilities of its base system. This approach (based on Gall's Law - http://principles-wiki.net/principles:gall_s_law), combined with the simplicity of DGP's architecture, significantly reduces the amount of net-new code needed to build each new software system, which is how it is able to reduce the amount of time and total cost required to deliver those systems.

DGPDrive is a reference system built by extending DGP which provides some useful file sharing and collaboration features, with very strong security, plus fine-grained control over which accounts are able to access specific functionality and content within the system. In addition, its open-source code and open documentation act as working examples showing how to build new software systems as extensions of the DGP base system.

DGP itself evolved over many years as a way to meet the requirements of the R&D groups in several large companies. The basic process followed by the R&D groups was to build functional prototypes which proved (or disproved) that new products and/or services were technically feasible before investing significant time and money into them. The overall process both starts and ends with documented requirements, which are first used to define what is to be designed and built, and are subsequently used again during testing to prove/disprove that the deliverables are in fact able to meet all of the requirements. However, attempting to build each functional prototype as its own individual software system took too much time, and resulted in a growing backlog of R&D prototypes waiting to be designed and built. A different approach was needed in order to deliver the large number of functional prototypes as quickly and inexpensively as possible.

As a first step, a very extensible general-purpose architecture was created that could be used as a foundation to build virtually any type of business software system. Second, a base system was built that was able to meet all of the combined NFR's for strong security, high performance and efficiency, good scalability, 100% system uptime, etc., which also included all of the security UIs, logging, monitoring, and other functionality needed by all business software systems. Third, the architecture and base system were

modified to make it very quick and easy to build new systems (prototypes) as extensions added to the base system, reusing all of its UIs and functionality instead of building each prototype as its own separate software system. This non-standard architecture and methodology proved to be very successful in practice. Subsequently, the base system was itself extended to enable it to be used for production systems instead of only for functional prototypes.

In recent years it has become evident that current software development methodologies and architectures typically deliver low-quality software systems with poor security. In order to verify that these observed problems are in fact 1) real, and 2) are the cause of significant negative effects, a collection of reports by CISQ, OWASP and other sources have been included as part of the DGP documentation. As one example, according to the CISQ 2022 report, the total cost to US businesses of low-quality software with poor security for 2022 has increased to $2.41 Trillion, with an additional $1.52 Trillion in technical debt. The combined costs of these problems represent almost 17% of the $23.3 Trillion US GDP for 2022. Even worse, those costs continue to increase every year.

What is the Purpose of DGP (Distributed Grid Platform)?

DGP's original purpose was to provide the same advanced features, functionality and capabilities as the big public cloud vendors for smaller, privately hosted custom software systems, using the much simpler architecture, code and methodologies of the R&D group universal architecture as its foundation. An abbreviated list of the NFR's include:

- Strong, effective security (able to pass PCI audits if necessary)
- High performance and efficiency
- True 100% system uptime, with no maintenance windows (no periods of planned system downtime for patching, etc.)
- Easy scalability, from very small to very large systems
- Easy system evolution (the ability to incrementally rewrite a software system while it remains in constant use)
- Zero code maintenance
- Continuous testing, with easy ad hoc, full regression and load testing
- Low total cost and short delivery timelines

Later, it was noticed that DGP's non-standard architecture, methodologies, etc. effectively solved virtually all of the problems that have been documented in the reports listed above. In addition, DGP provides the same advantages (reduced development costs and shorter delivery times) promised by low-code development platforms while using standard programming languages and development tools that are already known and are in current use.

Therefore, the purpose of DGP and DGPDrive can be summarized as follows:

1. Solve the technical problems that result in the delivery of low-quality software systems with poor security that have been documented in the various reports listed above.
2. Introduce a software development methodology based on the scientific method that is able to continuously prove how well (or how poorly) software systems are able to meet all of their requirements so that these problems are not repeated in the future.
3. Significantly reduce the total cost of software development and the amount of time it takes to deliver high-quality results.
4. Release all of the above as open-source, with open documentation, to enable these solutions and methodologies to be spread and used as widely as possible, at no cost.

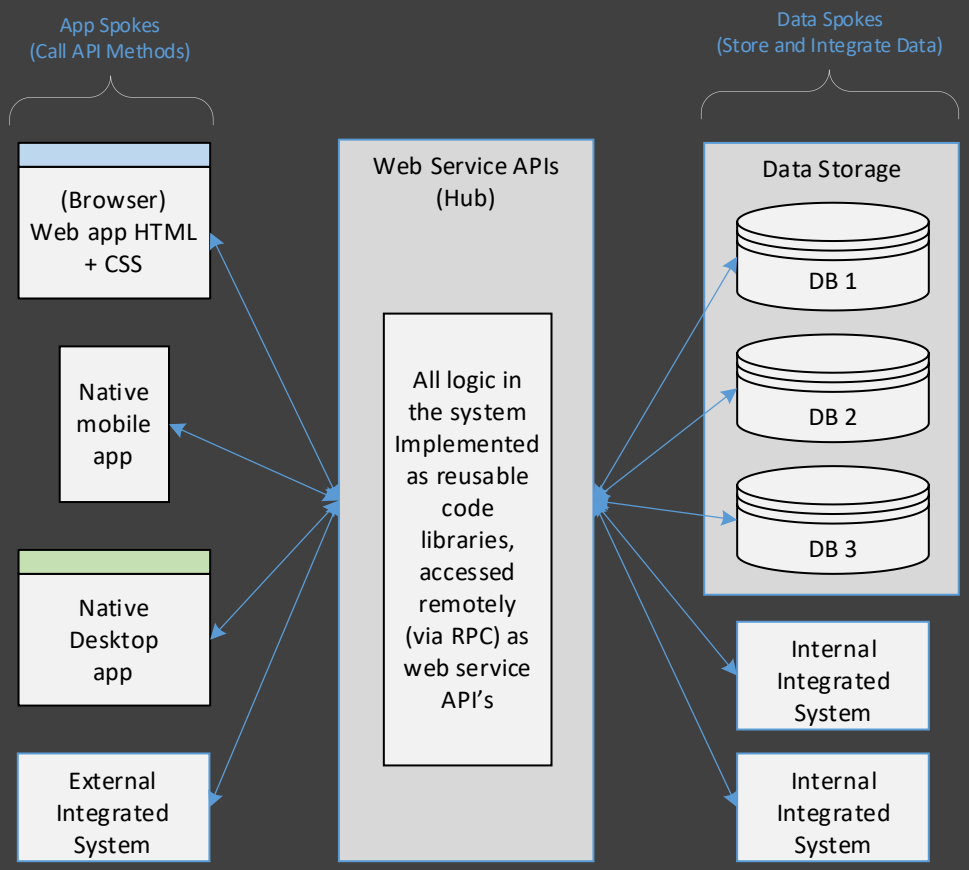
"To bear and not to own; to act and not lay claim; to do the work and let it go: for just letting it go is what makes it stay." — Lao Tzu

(URL's for most of the original reports included in the DGP documentation web app)

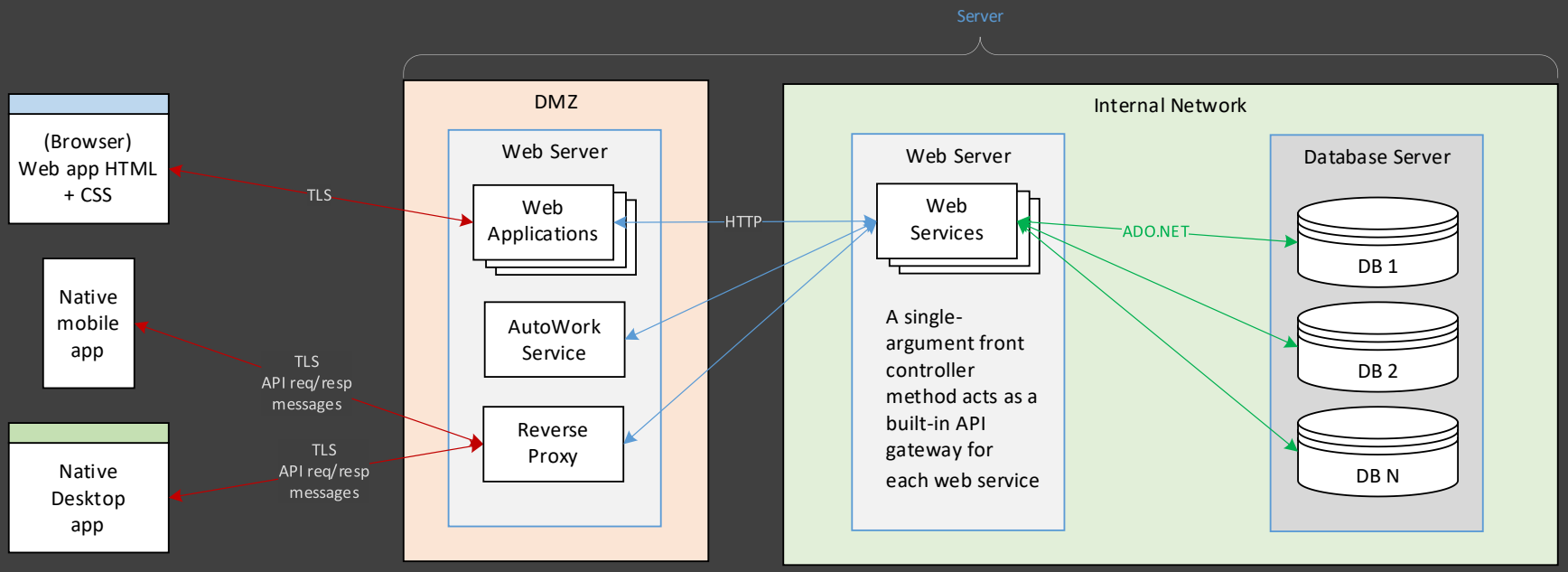
- CISQ "The Cost of Poor Software Quality in the US - 2022": https://www.it-cisq.org/technical-reports/
- CISQ "The Cost of Poor Software Quality in the US - 2020": https://www.it-cisq.org/technical-reports/
- OWASP Top 10: https://owasp.org/www-project-top-ten/
- OWASP API Top 10 https://owasp.org/www-project-api-security/
- Synopsis "Open-Source Security and Risk Analysis - 2023": https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2023.pdf
- Cyentia "State of Application Exploits in Security Incidents: https://www.f5.com/content/dam/f5-labs-v2/article/articles/reports/20210720_soso/The-State-of-the-State-of-Application-Exploits-in-Security-Incident-F5Labs.pdf

DGP Architecture and Design

At a high level, DGP's universal architecture is based on a simple hub-and-spoke design (shown below), in which all logic in a system is consolidated into a centralized hub, which is built as web service APIs.



The “spokes” on the left side of the diagram represent applications and other systems that use the web service APIs and depend on their functionality. Spokes on the right side of the diagram represent data storage subsystems that are used by the logic of the APIs in the centralized hub. The web service APIs are themselves very modular, similar to the modularity of microservices, for example.



The logical tiers of the hub-and-spoke design are then mapped to a two-tiered thin client/fat server topology, with both the logic (APIs) and the data spokes grouped together within the server tier (shown in the diagram above as the Internal Network). This simple architecture is extremely extensible, largely due to the loose coupling provided by the RPC’s that are used for communication between the tiers, implemented as configurable endpoints (URL’s, UNC paths, ADO.NET connection strings, etc.).

This allows new spokes and corresponding modular APIs to be added to a system almost indefinitely, while reusing all of the functionality of the base system while also reusing the growing amount of API logic in the system as well.

DGP's architecture has been proven to meet all of the NFR's listed above for the production systems that were built at those businesses at the time. The DGPDrive reference implementation is basically an open-source rewrite of those earlier production systems. This centralized thin client/fat server architecture provides the best overall security for business software systems, and also greatly simplifies their maintenance.

The client applications and data storage "spokes" are designed by default to contain no logic, but this restriction can be relaxed as needed. This simple API-centric architecture has been proven to deliver good results as long as the end-to-end performance of the web service API methods are very fast, efficient, scalable and able to provide strong, effective security. The internal network consisting of the logic hub (APIs) and data spokes is itself a modular multi-tier architecture that easily accommodates the ever-growing collection of APIs appended as extensions to a base system over time.

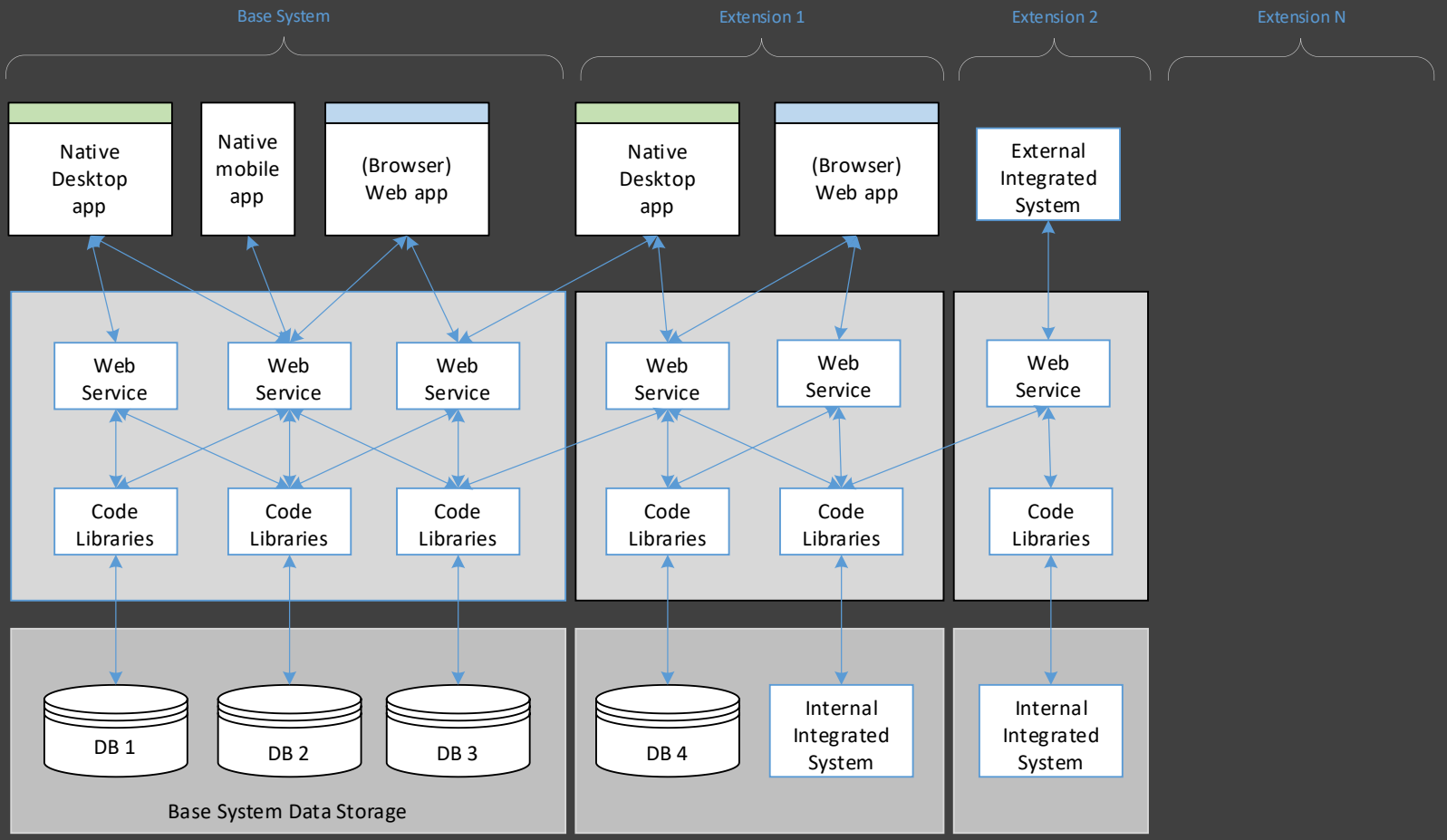
The overall process that is followed iteratively (and at different scales) is to first build a base system that delivers all the functionality needed to meet an organizations NFR's (security, performance, scalability, availability, etc.). This follows the concept of building large, complex systems by starting from a smaller, simpler working foundation.

It is very important to start new software systems from a small, simple, working foundation. Trying to build a large, complex system all at once is a much more difficult process that has a high chance of failure. The base system itself is designed and built following this same incremental concept, generally starting with the functionality of the data-driven RBAC security system, and then extending that core to incrementally meet all of the other NFR's (Gall's Law: https://en.wikipedia.org/wiki/John_Gall_(author))

The base system is then itself extended incrementally by following immutable append-only conventions, adding new APIs, code libraries, data storage, and applications while reusing all of the security, admin UIs, utilities, testing, logging, etc. from the base system. These extensions generally represent the functional requirements of specific products and services.

The combination of hub-and-spoke architecture, message-based APIs, the integrated data-driven RBAC security system, single-argument front controller web services, and the immutable append-only convention all work together to enable easy system evolution, competitive experimentation (prototypes), continuous testing, zero code maintenance, etc.

The diagram below illustrates how a hub-and-spoke base system is extended with new applications, APIs and data storage to implement new systems as extensions added to a base system, etc.



- System Evolution

For successful software systems, development continues throughout the entire useful life of the system, which can last for many years. Constantly appending new functionality and applications to a system without ever breaking backward compatibility is the core of DGP's system evolution, experimentation and zero code maintenance capabilities. At a high level, DGP's centralized APIs behave much like a server-based version of the .NET Framework, whose collection of APIs is shared by all the various "spokes" of the hub-and-spoke architecture. Each client app or integrated system only uses the specific API methods that they need from the ever-growing collection of APIs in a DGP system, in much the same way applications only use the desired namespaces in the .NET Framework. The data-driven security system controls which accounts are authorized to call which API methods. This not only enables experimentation, but also the incremental rewriting of a system to eliminate problems and technical debt, while it remains in constant use.

- Continuous Testing

This is one of the most important elements of Gall's Law: "Use automated testing to ensure that enhancing the system does not break it. Otherwise any further steps are enhancing of a non-working system". In addition, one of the cardinal rules of API-centric systems is to never cause breaking changes for the client applications that depend on the API functionality. Every deliverable must be tested and measured to prove that it is able to meet all of the documented requirements of a system, and just as importantly, to also prove that all of its functionality is working correctly at all times.

This change in emphasis from focusing on best practices and the development process to instead focusing on testing and measuring the deliverables produced by software development helps to significantly improve the quality of software systems. In DGP systems, this capability is based on the simple unit tests and performance measurements that are permanently built into every API method. The results of the unit test and server-side performance measurement are returned as part of every API response message, and can then be displayed to the users of the client applications. Plus, users that are members of the RemoteMonitor role automatically save these remote end-to-end test results and performance measurements to whichever location they are connected to at the time. This functionality is able to collect very useful real-time metrics from production systems without adversely affecting those systems due to the testing and monitoring workload. In addition, the API Tester test harness application is able to remotely run full end-to-end

regression tests of all of the API methods in a location. The test files used by the API Tester application contain a series of both positive and negative tests for each individual API method. The results of these end-to-end full regression tests can optionally be saved to the SysMetrics database for further analysis as desired. Finally, the API Tester application is also able to run load tests (synthetic workload) of a DGP location. Allowing many people to easily run the tests is part of the effort to ensure that the tests and measurements are independently verifiable by both the users and testers of each DGP system.

- Experimentation (functional prototypes)

The process of building APIs and/or API methods as prototypes (experiments) to either prove or disprove how well they are able to meet all of the requirements is really nothing more than using the normal system evolution capabilities to append new API methods to the code base of a system. The fine-grained authorization of the data-driven RBAC security system ensures that all new API methods automatically start out as prototypes which are only available to testers. The API Tester test harness and test files can then be used to fully test and measure both the functionality and performance of the prototype API methods in the standard way.

- Zero Code Maintenance

Another advantage of following the immutable append-only convention is the fact that no maintenance of existing DGP code is required. Once an API method is used in production it becomes immutable, along with its corresponding API Tester test file and documentation web page. From that point forward, no modifications to the API method (other than bug fixes) are permitted. The test files and documentation page for each API method are not strictly immutable, but once they have been completed to an acceptable level, they no longer require any modifications going forward since their respective API methods do not change. Ongoing enhancements, extensions and modifications to a system are implemented as new API methods or versions of methods that are appended to the code base instead of merging changes into any existing methods. Authorized access to these new methods is controlled by the RBAC security system in each environment as described above. Older methods can be deprecated once it is proven that no applications are using them. This mechanism is not only for system evolution, but is also used to eliminate technical debt, as new improved methods can be built to replace older methods that contain various types of technical limitations. The same type of immutable append-only convention also applies to database schemas, which are maintained using the DBSetup utility.

A very important side effect of zero code maintenance is that small teams of developers and testers are all that is ever needed for a software system, no matter how large it may grow over time. The team only needs to be large enough to keep up with the workload of adding new functionality to the system and testing it using the API Tester test harness application. All existing API methods are constantly tested as part of each end-to-end full regression to verify their continued correct functionality, and are never modified (or broken) by new development. The same is true for their associated test files and documentation pages. The combination of the zero maintenance and continuous testing innovations are responsible for the largest reductions in the total cost of software development for DGP systems, mainly due to the small size of development teams.

- 100% System Uptime and Automatic Recovery

The stateless web server farms used to host the web service APIs in each location of standard distributed systems provide very good fault-tolerance as long as the server farms contain some extra servers to handle the workload of any servers that go offline. Standard storage systems (RDBMS, NoSQL, etc.) used in distributed systems are able to replicate data between the nodes of a cluster within a single location, but they are generally not able to replicate data reliably between separate locations. However, the ideal functionality desired by businesses for mission-critical systems would allow multiple separate locations to be active and writeable at the same time in order to provide true 100% system uptime, which also effectively eliminates the need for Disaster Recovery and/or Business Continuity processes. The following URL provides a good overview of distributed systems, grid systems, their similarities and differences: https://www.toolbox.com/tech/cloud/articles/distributed-vs-grid/

The primary feature that differentiates a distributed grid from the more common distributed systems is having multiple locations that are active and writeable at the same time, along with automatic recovery of the system from failures of individual servers (including their data). These capabilities are the key to delivering true 100% system uptime, but the final critical requirement is the capability to patch and maintain all parts of a distributed system while the system remains in constant use (in other words, no maintenance windows of planned system downtime). Periods of planned downtime are not compatible with products and services that must be up and running 7 x 24 x 365. Delivering this functionality for the stateless web server farms hosting the web service APIs is relatively easy. Implementing that same functionality for data storage subsystems is the problem. Currently there are only 2 known options available that are able to do so (aside from the proprietary grid systems used internally by the big tech companies).

The first and currently the best option to deliver 100% system uptime is provided by SQL Server Availability Groups that have been configured to span across 2 separate datacenters. This technique has been used very successfully in production since 2014 by a credit card payment gateway vendor. This NFR for 100% system uptime is necessary because merchants with ecommerce websites need to be able to process credit card transactions 7 x 24 x 365, so any type of downtime for the payment gateway system is unacceptable.

The second option is DGP's omnidirectional data replication subsystem that has been specifically designed to continuously replicate data between multiple active, writable locations. By default, DGP's omnidirectional replication provides eventual data consistency between the multiple locations. However, strong data consistency can be achieved by designating one DGP location as the primary, used for all reads and writes in a system. This is essentially the same concept of a single writeable node borrowed from database clusters (including SQL Server Availability Groups) and applied to DGP locations.

DGP's data replication also supports a type of horizontal data scalability referred to as database shards. However, not all software systems need 100% uptime and all the redundant hardware and software required to deliver that level of availability, which is why the functionality to deliver 100% uptime are optional. Standard database or NoSQL clusters, etc. can be used for data storage whenever that is the best fit for a given set of requirements.

DGP/DGPDrive Beta SBOM (Software Bill of Materials)

- [closed source] Windows operating system.
  - Windows 10 (or later) for software development and Windows Server 2016 (or later) Standard Edition for production.
  - Windows IIS 10.x web server (built into Windows operating systems).
  - .NET Framework 4.8.1 (built into Windows operating systems).
- [closed source] SQL Server RDBMS.
  - SQL Server Express 2016 (or later) for Dev, Test environments
  - SQL Server Standard Edition 2016 (or later) for QA and Production environments.

Neither DGP nor DGPDrive use any open-source libraries, NuGet packages, tools or frameworks other than the dependencies in the SBOM list above. This means that all parts of DGP use functionality that is built into the Windows operating system. Since no open-source tools are used, no scans of their respective trees of dependencies for malicious code are necessary.

100% Self Support

The objective for users of DGP/DGPDrive's free open-source code and open documentation is to become 100% fully self-supporting, which effectively eliminates the need for any sort of external support for DGPDrive. The fact that a single individual (Alan Rahn) has been able to design, build, test, document and maintain DGP/DGPDrive is the best possible proof of the feasibility for small teams to be able to handle a subset of that same workload in order to become fully self-supporting in reality.

100% self-support is a realistic objective for the following reasons:

- The simplicity of the architecture and code. An entire low-level DGP system diagram can be shown on a single page.
- The small amount of simple code. All of the API functionality in a DGPDrive system totals approximately 12,000 lines of code. In addition, a majority of this code is the repetition of a few basic templates (mapper classes, data classes, and so on).
- The full documentation of DGP's architecture, design, configuration, deployment, testing and all of the other parts of DGP have all been built into the open-source code as a documentation web app included in every release.
- The immutable append-only conventions followed in all tiers of DGP which never alters any existing code, releases, or core data. This results in zero maintenance for all existing immutable APIs, API test files, and method documentation pages in each system.
- The testing and performance measurement permanently built into every DGP API method provide continuous testing.
- The full regression tests run on all API methods using the built-in API Tester test harness, which can also be used for load tests.

The ability for everyone to fully understand all of the relatively small amount of source code, plus all aspects of the system's design and implementation are necessary to eliminate the risk of depending on any one person for support going forward. Basically, everyone can support their own systems as if they had designed and built all parts of the system for themselves.

Summary

The CISQ 2022 report estimates the annual cost of poor-quality software to US business to be $2.41 trillion for 2022 in the US alone, with another $1.52 trillion in technical debt. Combined, that is almost 17% of the estimated 2022 US GDP of $23.35 trillion. It is also evident by comparison to a similar report from 2020 that these costs continue to increase at a significant pace, especially cybercrime, OSS supply chain security problems, and technical debt.

In order to fix all of these problems, while also reducing total costs and delivery timelines, DGP uses a development methodology that was created and refined over many years in the R&D groups of several large companies. It is based on the scientific method and focuses on the rigorous testing and measurement of the deliverables produced by each iteration of development in order to prove that they are able to meet all of the documented requirements. Finally, the data from the various tests and measurements must themselves be independently verifiable in order to catch any mistakes that may have been made during the overall process.

Most importantly, capabilities such as zero code maintenance + system evolution + continuous testing + the RBAC security subsystem significantly improves the way software systems are designed and built. The fastest way to build a new DGP software system is to append new functionality as extensions to an existing system. In terms of code maintenance, one of the biggest improvements is that it enables software systems to be incrementally rewritten in order to constantly add new functionality, correct mistakes and remove technical debt, all performed without breaking backward compatibility, while the system itself remains in constant use. These capabilities protect the investments made in custom software systems, plus this high level of adaptability can extend the useful productive lives of those systems.

Finally, the combination of open-source code, the simplicity of the code, the small amount of code and the detailed documentation of DGP's architecture, design, etc. included in each release all help to enable organizations to become truly 100% self-supporting in practice. The emphasis on self-support is intended to eliminate the risks of using a system that has been designed, built, tested, documented and maintained by a single individual (Alan Rahn).