Best Practice: Application Frameworks

CHRIS NOKLEBERG AND BRAD HAWKES, GOOGLE

WHILE POWERFUL, FRAMEWORKS ARE NOT FOR EVERYONE.

Shared libraries encourage code reuse, promote consistency across teams, and ultimately improve product velocity and quality. But application developers are still left to choose the right libraries to use, figure out how to configure those libraries correctly, and wire everything together. By preinstalling and preconfiguring libraries, application frameworks provide a simplified developer experience and even greater consistency, albeit at the cost of some flexibility.

By owning the entire application life cycle, frameworks go beyond a mere collection of libraries. Guaranteed framework behavior can scale development—for example, by avoiding the need for in-depth security or privacy code reviews of every application. The cross-team and cross-language consistency provided by frameworks is also a necessary foundation for higher-level automation and smart systems.

This article begins with an overview of the central aspects of frameworks, then dives deeper into the benefits of frameworks, the tradeoffs they entail, and the most important features we recommend implementing. Finally, the article presents a practical application of frameworks at Google: how developing a microservices platform allowed Google to break up its monolithic code base, and how frameworks enabled that change.

WHAT IS A FRAMEWORK?

A framework is in many ways similar to a shared library and has similar benefits. For Google, two technical principles help to distinguish a framework from a library: inversion of control and extensibility. While seemingly modest, the many benefits of frameworks discussed in this article are principally derived from these principles.

Inversion of control

In an application built from scratch, the engineer dictates the flow of the program—this is *normal control flow*. In a framework-based application, the framework controls the flow and will call into the user code—this is *inverted control flow*. Inverted control flow is sometimes referred to as the Hollywood principle: "Don't call us; we'll call you." The framework control flow is well defined and standard across all applications. Ideally, applications implement only the application-specific logic, and the framework can handle all of the other minutiae of building something like a microservice.

Extensibility

Extensibility is the second key differentiator from a library and goes hand in hand with inversion of control. Because the control flow of a framework is owned by the framework, the only mechanism to alter the framework's behavior is via the extension points it exposes. For example, a server framework might have an extension point that allows an application to run some code after every request. This behavior also implies that nonextensible parts of a framework are fixed and cannot be changed by applications.

BENEFITS OF FRAMEWORKS

Frameworks have multiple benefits beyond the functionality that shared libraries provide, and they are advantageous to a variety of stakeholders in different ways.

For developers

Developers, who ultimately decide whether or not to use an available framework, are the most obvious beneficiaries of frameworks. Primary developer benefits include increased productivity, simplicity, and conformance to best practices. Developers can write less code by leveraging built-in framework features, and the code they do write can be simpler because the framework handles boilerplate code. A framework provides a well-lit path for best practices by providing sensible defaults and eliminating pointless and time-consuming decision-making.

For product teams

In addition to improving developer productivity, frameworks benefit product teams by freeing up team resources that would otherwise be spent building redundant infrastructure. Product teams can then focus on what makes their product special.

Product teams also benefit when frameworks isolate them from changes in the underlying infrastructure. While not possible in all cases, the additional abstractions provided by a framework mean that some infrastructure migrations can be treated as implementation details that are handled entirely by whoever maintains the framework.

Product launches at Google often require signoff from multiple teams. For example, a launch coordination engineer is responsible for reviewing launches for production safety and effectiveness, while an information security engineer will check an application's design for common security vulnerabilities. A framework can simplify the launch-review process when the teams performing reviews are familiar with the frameworks and can rely on their behavioral guarantees. After launch, standardization will also make the system easier to manage.

For the company

At the company level, common frameworks can increase developer mobility by reducing how long it takes for developers to get up to speed on a new application. If a company has a sufficiently large community of developers, investing in high-quality documentation and training programs is worthwhile; this in turn helps attract documentation and code contributions from the community itself. Widespread usage of a framework also means that a relatively small investment in improvements to the framework can have a large impact.

Over time, centralization in framework architectures can allow widescale reaction to changing landscapes. For example, if you rely on a consistent microservice/

RPC (remote procedure call) framework and bandwidth becomes more expensive relative to CPU, then the framework defaults can adjust compression parameters centrally based on that cost tradeoff.

TRADEOFFS FOR FRAMEWORKS

While frameworks come with the multiple benefits just described, they also entail certain tradeoffs.

Opinionated frameworks can hinder innovation

Frameworks often have to make choices about which types of technologies to support. While supporting every conceivable technology is not practical, there are clear benefits when frameworks are *opinionated*—that is, when they encourage the use of some technologies or design patterns over others.

Opinionated frameworks can greatly simplify the job of developers approaching their new system with a blank slate. When developers have many ways to accomplish the same task, they can easily get buried in the details of decisions that have negligible impacts on the overall system. For these developers, accepting framework-preferred technologies of an opinionated framework allows them to focus on the business of building their system. Having a common and consistent preference also benefits the entire company, even if that answer is less than perfect.

Of course, you may have to deal with a long tail of applications and teams, and some product requirements or team preferences may not be well suited for existing frameworks. The framework maintainers are put in the position of deciding what is and isn't a best practice, and whether an unconventional use case is "valid" or not, which can be uncomfortable for everyone involved.

Another important consideration is that even if something is clearly a best practice today, technology evolves quickly, and there's a risk that frameworks will not keep pace with innovation. Experimenting with alternative application designs may be more expensive because developers need either to learn framework implementation details or to rely on assistance from framework maintainers.

Universality can lead to unnecessary abstractions

Many framework benefits such as common control surfaces (explained later) are realized only when a critical mass of applications use the same framework. Such a framework must be sufficiently generic to support the vast majority of use cases, which in practical terms means having a rich request life cycle and all the extensibility hooks that any application would need. These requirements necessarily add some layers of indirection between the application and underlying libraries, which can add both cognitive and CPU overhead. For application developers, more layers in the software stack can complicate debugging.

Another potential drawback of frameworks is that they are yet one more thing engineers have to learn. Newly hired Googlers are frequently overwhelmed by the number of technologies they need to learn just to get a "hello world" example working. A full-featured framework might make the situation worse, not better.

Google has mitigated these issues somewhat by trying to make the core of each framework as simple and performant as possible, and leaving other features as optional modules. Google also tries to provide framework-aware tools that can leverage the inherent structure of the frameworks to simplify debugging. Ultimately, however, frameworks have a cost that you must acknowledge, and you need to make sure that any given framework provides enough benefits to justify this cost. Different programming-language frameworks may also have differing sets of tradeoffs, creating another decision point and cost/benefit scenario for developers.

IMPORTANT FRAMEWORK FEATURES

As already discussed, inversion of control and extensibility are fundamental aspects of frameworks. Beyond those basic parameters, frameworks should account for several other features.

Standardized application life cycle

To reiterate, inversion of control means that a framework owns and standardizes an application's overall life cycle, but what benefit does this structure actually buy? The scenario of avoiding cascading failure provides one example.

Cascading failure is a well-known cause of system outages, including many at Google. It can occur when part of a distributed system fails, which then increases the probability that other parts will fail. For more information on the causes of cascading failures and how to avoid them, see the chapter on Addressing Cascading Failures in *Site*

Reliability Engineering (O'Reilly Media, 2016).

Server frameworks at Google have a number of built-in protections against cascading failure. Two of the most important principles are:

- ***Keep serving.*** If a server can answer requests successfully, it should do so. If it can successfully serve some kinds of requests but not other kinds, it should continue running and answer the requests that it can serve.
- ***Start up quickly.*** The server should start up as quickly as possible. Faster startup means faster recovery from crashes. The server should avoid waiting serially for initializations involving RPCs to external systems to complete.

The Google production environment gives each server a configurable amount of time to become "healthy" (start responding to requests). If the time expires, the system assumes that an unrecoverable error occurred and terminates the server process.

There is one common antipattern that occurs naturally in the absence of a framework: A library creates its own RPC connection and then waits for that connection to be ready. As a server code base grows over time, you can end up with literally dozens of such libraries in the transitive dependencies. The result is server initialization code, which if unrolled effectively looks like figure 1.

Under normal circumstances, this code will work fine, which is especially problematic because there is no indication of a lurking problem. That problem shows itself when one of the associated back-end services slows down or goes down altogether—now the primary server's

FIGURE 1: **SERVER INITIALIZATION CODE**

```
// In library 1
// newStub creates a stub which will asynchronously connect to a backend.
FooService.Stub fooStub = FooService.newStub(...);
// waitUntilReady blocks until the stub is successfully connected.
waitUntilReady(fooStub);
// In library 2
BarService.Stub barStub = BarService.newStub(...);
waitUntilReady(barStub);
// In library 3
BazService.Stub bazStub = BazService.newStub(...);
waitUntilReady(bazStub);
```

startup is delayed. If the startup is sufficiently delayed, it will be killed before it ever gets a chance to start handling requests, which can contribute to a cascading failure.

One possible improvement is to create the RPC stubs first, as in figure 2, and then wait for them all in parallel. In this scenario, you need to wait only for the *max* of the stub initialization times rather than the *sum*.

While still not perfect, even this limited refactoring demonstrates that you need *some* coordination between the libraries creating the RPC stubs—they must hand off responsibility for waiting for the stub to something outside of the library. In Google's case, that responsibility is owned by the server framework, which also has the following features:

- Waiting for all stubs to be ready *in parallel*, by polling readiness periodically (< 1 sec). Once a configurable

FIGURE 2: **RPC STUBS**

```
// Make sure to call newStub for all stubs first, before we wait for any of
them.
FooService.Stub fooStub = FooService.newStub(...); // In library 1
BarService.Stub barStub = BarService.newStub(...); // In library 2
BazService.Stub bazStub = BazService.newStub(...); // In library 3
// Wait, now that we have started the async connection process for all
stubs.
// The order in which we wait for the stubs is irrelevant.
waitUntilReady(fooStub);
waitUntilReady(barStub);
waitUntilReady(bazStub);
```

timeout has elapsed, the server can continue with initialization *even if not all back ends are ready.*

- Emitting human- and machine-readable logging for debugging and integration with standard monitoring and alerting systems.
- Plugging in arbitrary resources, not just RPC stubs, through a generic mechanism. Technically, only a function returning a Boolean (for "Am I ready?") and a name is necessary for logging purposes. These hooks are typically used by the common libraries that deal with resources (e.g., a file API); application developers often get the behavior automatically just by using the library.
- Providing a centralized way to configure certain back ends as "critical," which alters their startup and runtime behavior.

These features would (rightly) be considered overkill for any individual library, but implementing them makes sense if you can do so in a central place from which all back-end-using libraries can benefit. Just as shared libraries are a way to share code among applications, in this case the framework is a way for libraries themselves to share functionality.

SREs (site reliability engineers) are much happier to support framework-based servers because of features such as these, and they often encourage their developer counterparts to choose framework-based solutions. Frameworks provide a baseline level of production regularity that is difficult—if not impossible—to achieve when just gluing together a bunch of disconnected libraries.

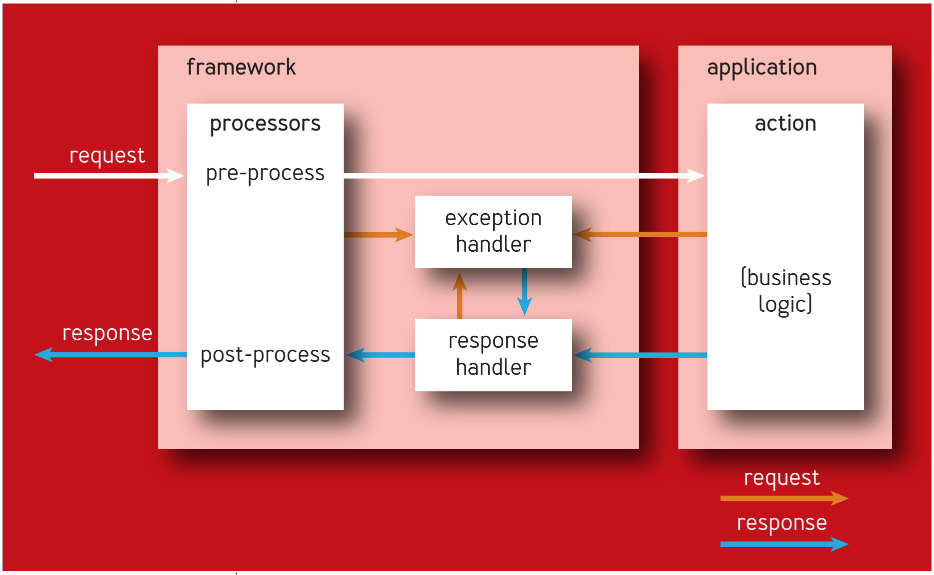
Standardized request life cycle

While details vary depending on the type of application, many frameworks support additional life cycles beyond an overall application life cycle. For Google server frameworks, the most important unit of work is a request. Following a similar inversion-of-control model, the goal of the request life cycle is to divide the responsibilities for different aspects of the request into separate extensible pieces of code. This allows application developers to concentrate on writing the actual business logic that makes their application unique.

Here's an example of one such real-world framework and its component pieces, as illustrated in figure 3:

- *Processors*—intercept incoming and outgoing payloads. Mostly used for logging but have some capabilities for short-circuiting a request (e.g., enforcing invariants across an entire application).

FIGURE 3: **AN EXAMPLE FRAMEWORK AND ITS COMPONENT PIECES**



- *Action*–application business logic that takes the request and returns a response object, possibly with side effects.
- *Exception handler*–converts an uncaught exception into a response object.
- *Response handler*–serializes a response object to the client.

While applications can take advantage of the framework extension points, the vast majority of application code takes the form of actions, which embody the application-specific business logic.

This separation of concerns has been helpful in the realm of web security, for example. Google develops many web applications, so has a strong desire to guard against all of the various web security vulnerabilities, such as

XSS (cross-site scripting). XSS vulnerabilities are often caused by application code returning a string response containing insufficiently sanitized or escaped data. The traditional approach to fixing these bugs is simply to add in the missing escaping data and hope that tests and code reviews will prevent similar problems in the future (spoiler: they will not).

Fundamentally, this approach won't work because the underlying APIs that the applications are coding against are inherently prone to bugs like XSS because they accept strings or similarly unstructured/untyped data. For example, the Java Servlet API gives applications a raw Writer, to which you can pass arbitrary characters. This approach puts too much of a burden on developers to do the right thing; instead, the security team at Google has focused on designing inherently safe APIs, such as:

- HTML template systems with contextual-aware escaping.
- SQL-injection-resistant database APIs.
- "Safe HTML" wrapper types that carry contracts stipulating that their value is safe to use in various contexts.

The request life cycle of Google's server frameworks complements the use of these APIs because the application code never deals with raw strings or bytes. Instead, the code returns high-level response objects with types such as `SafeHtmlResponse` that can be constructed only in ways that are guaranteed to be well formed. Turning those response objects into bytes on the wire is the responsibility of a *response handler*, which is typically a built-in part of the framework. Google sometimes needs custom response handlers, but all usages must be reviewed by the security team—a requirement that is enforced at the build level.

The net impact is that Google has reduced the number of XSS vulnerabilities in applications using these frameworks to virtually zero. As you can imagine, Google's security team strongly encourages the use of frameworks and has made many framework contributions to improve the security story for all framework users. A standard framework-based server can effectively skip many of the security or privacy reviews that a bespoke server would require to launch, since the framework is trusted to guarantee certain behaviors.

Of course, the benefits of a structured and extensible request life cycle go well beyond separating business logic from response serialization. The most basic benefit is that it keeps each component small and easy to reason about, which helps long-term code health. Other infrastructure teams within Google can easily extend framework functionality without working directly with the framework team. Finally, applications can introduce their own cross-cutting features without touching each action. In some cases, these features are domain specific, but other features end up being generally applicable and are eventually "upstreamed" into the framework itself.

Common control surfaces

What we call *control surfaces* include all of the non-application-specific inputs and outputs of a binary when viewed as a black box. These include operational controls, monitoring, logging, and configuration.

Uniformity across servers makes troubleshooting problems much easier. Regardless of which server they're troubleshooting, developers and SREs all know what information is available and where to look for it. If something about a server needs to be tweaked, everyone knows which knobs are available and how to change them.

Beyond making it easier for humans to operate servers, having common control surfaces across servers also makes shared automation viable. For example, if all servers export errors in a standard way, changing the release pipeline to perform automatic canarying becomes possible: You can first roll out a new binary to a few servers and look for a spike in errors before performing a wider rollout. You can read more about the benefits of a common control surface from the SRE's perspective in the chapter on the Evolving SRE Engagement Model in *Site Reliability Engineering* (O'Reilly, 2016).

Frameworks provide a great opportunity to enforce a level of uniformity across application control surfaces. While, generally, only a few people care about the exact composition of control surfaces, there is tremendous value to the company in having a single consistent answer. Consistency means that you can easily share and scale automation across multiple binaries. By simplifying integration with the surrounding ecosystem, you can reap the benefits of *having* a standard many times over, independent of the merits of the standard itself.

One challenge of implementing a common control surface is that framework maintainers are often the first to discover inconsistencies across programming-language libraries. For example, all languages had an existing notion of a command-line argument whose value was a duration (a length of time). On the positive side, the syntax was somewhat compatible among languages, at least for the most basic examples such as "1h30m." Once we dug into the details, however, a different picture emerged, as shown in figure 4.

These days, library owners have a greater awareness of the value of cross-language consistency and the need to take such consistency into consideration. On the framework side, Google also uses test harnesses to run the same suite of tests against servers written in each programming language to ensure consistency going forward.

Modularity

FIGURE 4: **INCONSISTENCIES ACROSS PROGRAMMING LANGUAGE LIBRARIES**

	C++	JAVA
Days ("d")		✓
Hours ("h")	✓	✓
Minutes ("m")	✓	✓
Seconds ("s")	✓	✓
Milliseconds ("ms")	✓	
Microseconds ("us")	✓	
Nanoseconds ("ns")	✓	
Units out of order	✓	
Repeated units	✓	
Fractional values	✓	
Unitless value		✓
Mixed case		✓

For better or worse, there is no central software engineering authority at Google. Although most developers work against a single code repository, engineering practices still vary significantly among teams. The choice of technologies for any given project typically rests with the tech lead of the project, with few top-down mandates. Understandably, people tend to choose technologies with which they have prior experience. As a result, in order for a new technology to gain significant adoption, it must have either an obvious value or a low barrier to entry; typically, it must have both.

For Google's server frameworks, core life-cycle management and request dispatching are the only strictly required features. All other functionality is bundled into optional, independent "modules" that implement their functionality using the various life-cycle hooks exposed by the framework, as discussed earlier. Application developers can pick and choose which modules to add to their server, and in many cases even major features can be added via a one-liner:

```
install(new LoadSheddingModule());
```

The actual list of available standard modules numbers in the hundreds, including features such as authentication, experiments, and logging.

The ability to add framework features incrementally to a server was a big factor in framework adoption at Google. It allowed for "hello world" examples and prototype servers to be small and easy to understand, while still making it simple to scale up to a more full-featured server when appropriate.

The independence of the modules also allows for easy substitution of an application-specific module for a standard framework module if you have special requirements. Because standard framework modules use the same extensibility APIs as application-specific modules, upstreaming a useful feature into the framework is usually a trivial matter of moving code. This allows a framework to be an ever-growing collection of best practices, once these practices have proven their value in the real world.

The high degree of encapsulation exhibited here means that framework maintainers can radically change the implementation of a module without touching any application code. This is especially useful when a back-end system is deprecated or requires API changes (which is distressingly common). Google frameworks have insulated many application developers from needing to perform complex or costly migrations, and for many teams this is one of the most compelling benefits of frameworks today.

One role of a framework maintainer is to ensure that modules collaborate with each other correctly. Maintainers can also select default module lists or provide recommendations and constraints about which modules should be used for different situations. One challenge is striking the right balance with regard to granularity: While developers tend to prefer fine-grained modules for flexibility, it's harder for framework maintainers to ensure that all combinations will work well together.

MICROSERVICES

The standardization provided by widespread use of frameworks leads to opportunities for higher-level tools and automation. This allowed Google to create a microservices platform and break up the monolithic servers.

Before microservices: the monolith

The existence of shared libraries and frameworks has greatly simplified the actual act of writing production-quality code within Google. Writing code, however, is just one part of deploying an application at Google. Other critical ingredients include integration testing; launch reviews for aspects such as security and privacy; acquiring production resources; performing releases; collecting and saving logs; experimentation; and debugging and resolving outages.

Historically, handling all of these items was an expensive process that all server owners had to complete, regardless of the size of the server. As a result, instead of deploying new servers, smaller teams adding a new service would look for an *existing* server to which they could add their code. This way, the team could just focus on writing their business logic and get everything else "for free." Of course, once enough teams took this approach, it became clear that piggybacking onto existing server functionality was not actually free. This incentive structure resulted in a tragedy of the commons many times at Google: Well-supported servers continued to grow and grow until they became huge and unmaintainable monoliths.

Monoliths have many negative consequences. On the developer productivity front, you must deal with slow builds, slow server startup, and a high likelihood that your presubmit tests will break when you try to submit your change. For example, one important Google Search-related C++ binary became so large that it was impossible even to link, given technical limits at the time (12 GB RAM).

When it comes to releases, it's hard to push them to monoliths on schedule. As a monolith grows, so too does the number of contributing developers, which naturally results in more blocking bugs. A delayed release may make achieving the next release even more difficult, which can create a vicious cycle.

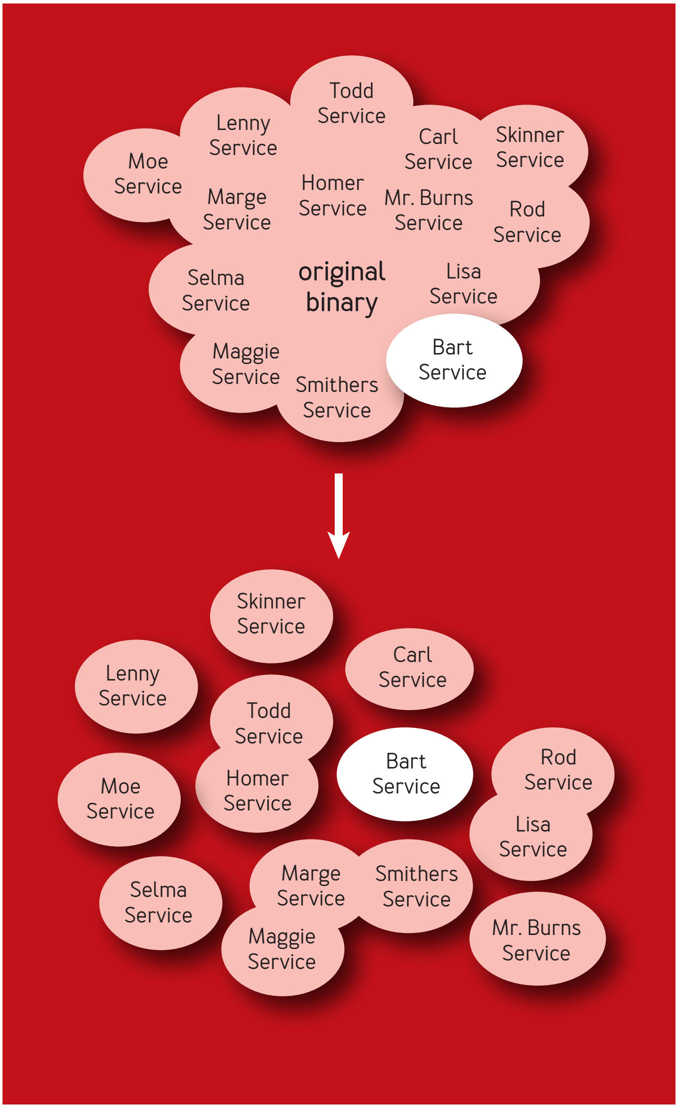
In production, monoliths create a dangerous shared fate between ostensibly unrelated services, as well as a greater chance of bugs caused by unexpected interactions. Scaling services independently of one another is impossible, which makes resource provisioning more difficult.

Moving away from a server-oriented world

Although it eventually became clear that the monolith situation at Google was unsustainable, there was no good alternative. Simply mandating that people stop adding to a monolith would have had equally bad consequences. Instead, Google needed to eliminate the toil of productionizing and running a new server. That would allow the decision of which *services* should make up a *server* to be based on purely production reasons, rather than developer convenience, shown in figure 5.

Working backward from the goal that developers should just focus on the business logic of their application-specific service, and that *everything else* should be

FIGURE 5: **BREAKING UP A MONOLITHIC SERVER INTO SMALLER SERVERS**



automated as much as possible, some requirements eventually became clear:

- Developers should be declaring and implementing service APIs, not writing `main` methods; orchestrating how a binary is actually run is the role of the microservice platform.
- All of the metadata needed for automation, including production configuration, should live in a declarative format alongside the code for the service.
- Resources and dependencies between services should be explicit and declarative. Ideally, you should be able to visualize the entire production topology just from looking at the metadata for the universe of services.
- Services should be isolated from each other, so that arbitrary services can be co-assembled into a server. Among other requirements, this means avoiding global state and side effects.

When these requirements are satisfied, virtually all formerly manual processes can be automated. For example, testing infrastructure can use the metadata to wire up a portion of the service-dependency graph when running integration tests.

A microservice platform using these principles was developed at Google, initially for the purpose of breaking up a particular large monolithic server that had seen rapid growth as a result of intense feature development. Once the platform proved beneficial, it was organically adopted by other teams within Google and was eventually spun out into a separate officially supported project.

Today the platform is the de facto standard for new server development, in part because it appeals to both small teams and large organizations. Because of the high level of automation, small teams can now easily turn up a Google-quality production service in a matter of days, whereas before a turn-up following best practices might have taken months. For large organizations, the consistency across teams reduces support costs, and the shared platform means that staffing org-specific infrastructure teams is often unnecessary.

Another benefit of moving to microservices has been encouraging developers to think more about the proper division of work among services, which has led to more rational system architectures. Using technologies such as gRPC and protocol buffers as the boundary between separate systems forces you to consider the APIs in a way that doesn't necessarily happen when you're only using function calls in the same process. RPC systems are also language agnostic, so each microservice owner can independently decide which language to use.

One remaining challenge, and a ripe area for future work, is providing higher-level tools to manage an ever-growing number of microservices. For example, monitoring consoles that were written in the previous era may have assumed a relatively small number of unique binaries, and this will require a new user interface to accommodate the much greater number of binaries that arises when people fully adopt microservices.

Relationship with frameworks

Frameworks are a critical component of making Google's microservices platform work for a few reasons:

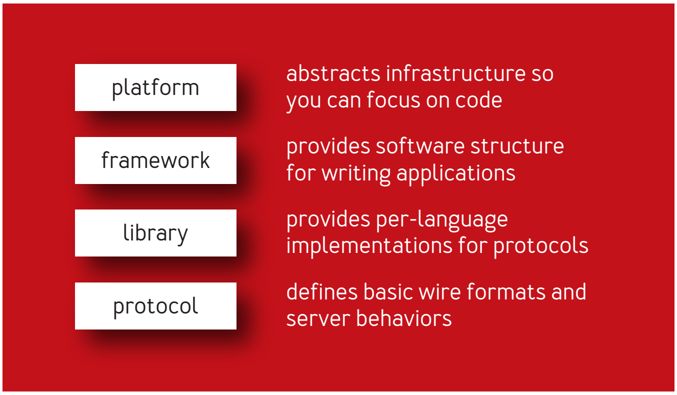
- The inversion of control inherent in the framework's life cycle naturally lends itself to a model where application developers just hand off their service implementation to the platform.

- Common control services (across both servers and languages) are required for many platform features, including release management, monitoring, and logging.
- Modularity means that both the platform and application code can provide independent modules, which when combined together, work in a sane way to form a complete server.

Figure 6 shows the full development stack for Google's microservices platform:

As discussed before, frameworks can offer a greater level of encapsulation than libraries, which simplifies writing applications and provides isolation from underlying library churn. In a similar way, the microservices platform goes beyond just code to encapsulate other artifacts such as production configuration. This allows for a

FIGURE 6: **DEVELOPMENT STACK OF GOOGLE'S MICROSERVICES PLATFORM**


platform
abstracts infrastructure so you can focus on code
framework
provides software structure for writing applications
library
provides per-language implementations for protocols
protocol
defines basic wire formats and server behaviors

corresponding higher level of simplification and isolation from churn. For example, the platform maintainers can (if necessary) automatically apply an emergency code fix or configuration change that rebuilds all affected binaries and pushes them to production in a uniform way—previously impossible.

Using a microservices platform, however, does present some challenges. One of the biggest of these is that enforcing all of the invariants required to make the microservices platform function properly can be onerous and may even affect how applications are coded. To provide one example, Google's Java servers share certain thread pools. Combined with the requirement that all services must be isolated from each other, this implies that a blocking thread-per-request model cannot be allowed—it would be too easy for a blocking service to use up all the threads and starve another service. For that reason, servers are mandated to be async only, a solution that not all teams are happy with.

Another challenge is that adding more hops between microservices may add latency to the overall request. In some cases, this latency can be mitigated by architectural improvements that happen as part of a microservices rewrite. For its microservices platform, Google has also ensured that requests between services that happen to be co-located in the same server use an optimized in-process transport.

CONCLUSION

While frameworks can be a powerful tool, they have some disadvantages and may not make sense for all organizations. Framework maintainers need to provide standardization and well-defined behavior while not being overly prescriptive. When frameworks strike the right balance, however, they can offer large developer productivity gains. The consistency provided by widespread use of frameworks is a boon for other teams such as SRE and security that have a vested interest in the quality of applications. Additionally, the structure of frameworks provides a foundation for building higher-level abstractions such as microservices platforms, which unlock new opportunities for system architecture and automation. At Google, such frameworks and platforms have seen broad organic adoption and have had a significant positive impact.

Brad Hawkes *is a senior software engineer working on core infrastructure at Google. He works on the server framework on the Java Virtual Machine, which is used in thousands of servers across Google. He is on LinkedIn at www.linkedin.com/in/bhawkes.*

Chris Nokleberg *is a principal software engineer and tech lead of server frameworks at Google. He started developing and evangelizing frameworks as a tech lead on Google Docs almost 10 years ago. His current focus is helping large teams adopt Google's microservices platform and standardizing developer best practices across the company.*