Script V2 Performance Report

Exploration Phase

Document Purpose

1. Provide detailed visibility into the performance characteristics of the Scripts v2 Platform.
2. Compare characteristics of an ***in process execution environment*** to an ***out of process execution environment*****,** and in turn, make a recommendation as to the execution architecture of Scripts v2.
3. Summarize upcoming performance optimizations that will drive latencies down even further with more investment.

TLDR: Conclusions

The conclusions herein with respect to the above purpose are:

1. Current platform end to end latency is sitting around **p95 = 25ms**. We have line of sight towards **p95 = 10ms** with upcoming optimization work. However, the significant (80% total time) performance work will be in optimizing how script input data can be loaded from core.
2. After prototyping an in-process Wasm runtime, we still recommend an out-of-process architecture due to stability/uptime risks and negligible differences in performance.
3. We have many options for continued improvement to drive latency further down.

Current Scripts v2 Workflows

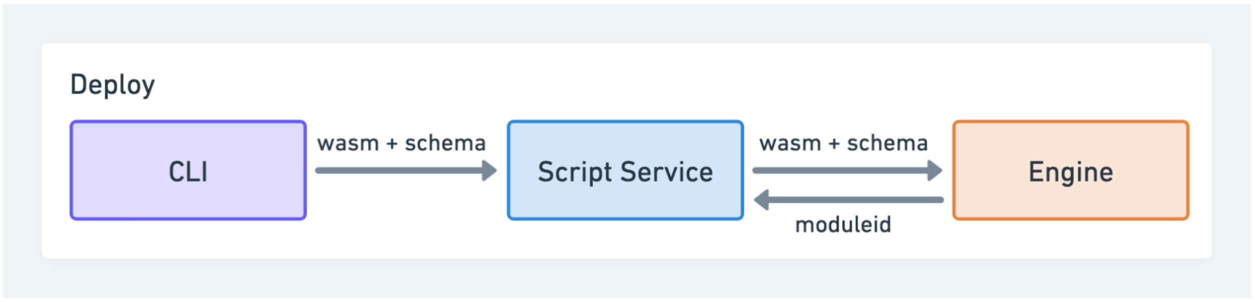
There are two core workflows in Script V2.

1. **Deploy**: A Script, written by a developer, is compiled to WebAssembly and stored in Shopify.
2. **Execute:** During some commerce workflow, a WebAssembly module is executed to provide real-time customization of that workflow.

Deploy Stage

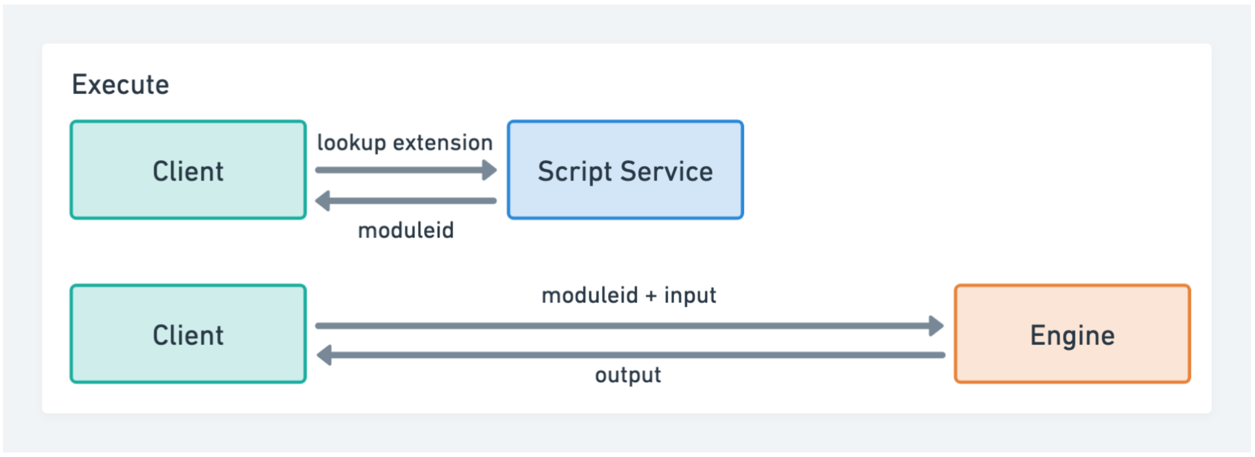
During the deploy stage, the Shopify App CLI compiles the user's code into a WebAssembly (Wasm) module and sends it along with the Extension Point (EP) schema to Script Service, which then forwards to Engine, where the Wasm module can be executed later.

Script Service sits in among all the services and manages the business logic, like binding the script to a specific app or a shop. Engine in this scenario serves as a WASM registry and execution environment. One thing to be noted: it doesn't store the WASM module directly, but actually AOT compiles the WASM module into an object file (x86 machine code), which gets executed faster in the next phase.



Execution Stage

During the Execution stage, the Runtime Client ruby gem first looks up the extension from Script Service and then sends the request to Engine to execute the pre-compiled WASM module with the input given by the call site.



When we talk about the performance, we mainly care about the Execution stage, given Deploy stage is an offline and ahead-of-time operation. Mentally, it’s similar to compilation and execution in a static language.

Baseline Performance

Steps of Execution

Currently, Scripts v2 is running using an out-of-process Wasm runtime environment called runtime-engine. We will first summarize the baseline performance of this solution, and then later go into results from the in-process prototype.

The execution of a v2 script is controlled by the Runtime Client gem. The execution is broken up into the following stages:

1. Look up extension
For a given EP name and scope value (shop id), figure out which Wasm module to run. This is a highly cached operation. It's broken down into more steps:

- **A. Request Extension:** Request the precompiled Wasm module id from the Script-Service. This part is cached.
- **B. Build Extension:** Convert the cached contents into a representation that can be executed (Ruby object creation from serialized cache contents).

2. Prepare input
Prepare the appropriate input that we can send to the Wasm module. This is broken down into a few sub-steps:

- A. **Load arguments:** The appropriate input is assembled as a Ruby hash, by the code at the call site. The Scripts infrastructure has no control over this code. *This code may involve running ActiveRecord queries to load the appropriate input.*
- B. **Validate arguments**: Validate the provided input against the extension's schema.
- C. **Serialize arguments:** Serialize input, so we can send it to the Wasm engine.

3. Run
Execute the Wasm module. This is further broken down into sections:

- A. **Time measured by client**: This would include any time on the network to access a remote Wasm engine.
- B. **Time measured by Wasm engine:** This would exclude any time in transit, and is the time measured by the Wasm engine to execute the module.

4. Callback
The EP author handles output, to apply the result of the Wasm module at the call site (such as applying a discount to the checkout). The Scripts infrastructure has no control over this code.

Baseline Performance Values

By load testing, we get the baseline performance. The following data tries to distinguish between blocks of code outside the Scripts platform's control (**user code**) and blocks of code inside Scripts platform's control (**runtime code**).

Sample discount script running at discount EP for 10 mins on May 1st. Dashboard here.

	p50 (ms)	*p95 (ms)*
Total	**64.49**	**112.28**

Total (user)	**53.62**	**95**
Total (runtime)	10.87	17.28
1. Look up extension (cached)	1.25	2.18
2. Prepare Inputs	12.31	25.12
a. load arguments (load the checkout object)	9.18	18.62
b. validate arguments	1.02	1.62
c. serialize arguments	2.11	4.88
3. Run	6.49	8.60
a. network	3.56	5.2
b. runtime-engine	2.93	3.4
4. Callback	44.44	76.38

Here is the breakdown inside runtime-engine.

As you can see, at least ~75% of the latency is spent between code that is gathering input for the script, and code that applies the output of the script. For this discount EP, the **load_arguments** method is producing a Ruby hash representation of the checkout. The **callback** is using the **Appliers::ItemExplicitDiscountApplier** class to apply the discounts to the checkout. We believe this is fairly representative of what Shopifolk might want to do with a Scripts EP, as this is quite similar to the input/output for line item scripts in scripts v1.

Goal

The goal we set up is to have **p95 of Scripts Platform latencies to be < 5ms**. This means the sum of latencies that are within the control of the Scripts platform (yellow highlight above). Currently we are sitting about 5x this goal. Where are the opportunities to improve this further?

What is Achievable?

The following section details, for each execution step, what further investments can be conducted to drive latencies down further.

When taking into consideration the following improvements, we'd be looking at a number much closer to **p95 = 8-10ms.** Not quite 5ms, but much closer. Once we get there, further analysis will inform how we reach the 5ms goal.

Look up extension (cached)
This latency is measuring a cache hit on memcached. Since the number of extensions will be very low relative to the number of times they are invoked, this data could be loaded into memory where the environment permits. For flash sales, this could be as easy as placing a memory LRU cache in front of memcached. For other shops, it could mean having a post-boot step in core to load the memcached contents redundantly into memory. In any case, the volume of data will be so low (using scripts v1 as a proxy), a memory cache solution should be feasible and should virtually remove this latency. *Improvements here should remove this latency from the total*.

Validate arguments
For this step we are considering a heuristics based approach where input validation is sampled, or perhaps removed altogether in production. Validation here is not strictly necessary and only serves to improve the error message provided back to Shopifolk (though arguably this should all be caught in development). *Improvements here should remove this latency from the total*.

Serialize Arguments
The current solution of using `Hash.as_json` could be much better. We could invest in other wire protocols (protobuf, thrift). Currently we serialize/deserialize data twice since the runtime-engine service has its own internal data structure for capturing inputs before writing bytes to the Wasm VM's linear memory. We could also implement a single serialization pass by having the Gem write the bytes out directly. *Improvements here should permit us to drive serialization time down at least 50%.*

Run: network
Currently we use HTTP 1.1 with a keepalive and multi-region deployment to ensure geographic proximity. We could explore more efficient protocols on top of HTTP/2. We could also work with production engineering to ensure our deployments are as close as possible to the pod that is invoking the script. Theoretically, there should be no reason we can't reduce network latency here to numbers seen when making a Memcached API call. *Improvements here should permit us to drive serialization time down at least 50%.*

Run: runtime-engine
Since this metric is currently not broken down by the guest module's code versus our code, it's hard to know where the opportunities live here. We know the extra serialization step could be improved, as per above. Further improvements to this section will be left out of scope for this section.

Performance Exploration

This section captures explorations done to answer two important questions:

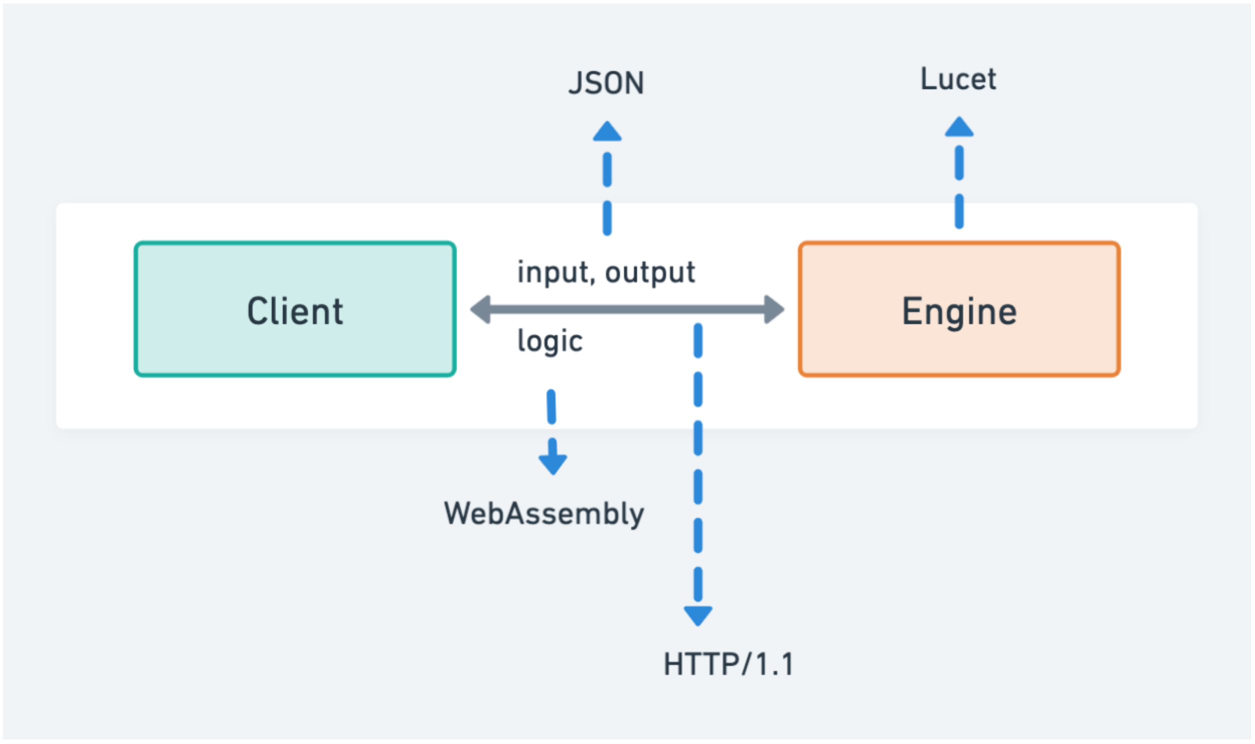
1. Which Wasm VM should we use to run Scripts?

2. Should Wasm modules at Shopify run in-process (right on core), or out-of-process (on the runtime-engine service)?

Client / Server Model

The extension run process follows the classic client-server model. Whether we are running in-process or out-of-process, the same core workflow applies. Only the details of the infrastructure changes, such as:

- Where the modules are stored? (ie. GCS vs DB vs Memcached)
- How do we serialize the input? (ie. JSON vs Binary)
- How do we pass the input to the Wasm VM? (ie. HTTP/1.1 or local FFI)



The logic (extension) is represented in WebAssembly, which features a secure and fast runtime.

Theoretically, we can send the WASM module (**.wasm** file) directly and let Engine execute it in JIT mode. Based on our benchmark metric, enabling the AOT compilation makes execution much faster, by having the Wasm VM first compile the **.wasm** file to an object file (**.so**), and storing the **.so** file for execution later. Therefore, we always want to AOT compile in the Deploy stage, so that in the Execute stage, we only have to fetch and run the **.so** file.

WebAssembly Runtime

Runtime-Engine is a wrapper over a WebAssembly runtime. According to our AOT use case and other research, we scoped down two candidates to run Shopify's Wasm: **Lucet** and **Wasmer** (with multiple backends). We conducted some benchmarks to see which is superior.

AOT compile (storage space)

	source	*wasmer/singlepass*	*wasmer/cranelift*	*wasmer/llvm*	*lucet*
add-one	2.1 K	41 K	18 K	13 K	21 K
nobody	9.3 K	222 K	62 K	30 K	58 K
fibonacci	16 K	362 K	98 K	43 K	86 K

AOT execution (latency)

	wasmer/singlepass	*wasmer/cranelift*	*wasmer/llvm*	*lucet*
add-one	244.24 us	89.810 us	904.55 us	175.81 us
nobody	1.2892 ms	202.61 us	2.1254 ms	205.35 us
fibonacci	1.9484 ms	221.65 us	2.0697 ms	194.29 us

Recommendation

Both Lucet (the one we are using) and Wasmer (Cranelift backend) have great numbers. **For now we will remain with Lucet, since we have already integrated it and since it's maintained by a strong team at Fastly. As the ecosystem evolves, we will re-evaluate this decision.**

Execution Architecture

Engine serves three purposes in our system today:

1. Compiles WASM modules (**.wasm**) to object files (**.so**).
2. Stores compiled WASM modules, acting as a WASM registry.
3. Executes compiled WASM modules, acting as a WASM runtime.

The current approach is Client sends the extension run request to Engine via HTTP/1.1, as shown in the baseline performance.

This is considered an out-of-process solution, since runtime-engine lives in a different process on a different machine. As a result, it incurs the network cost on sending data back and forth. As a counterpart, there should be an in-process solution that brings the Engine into the same process as the Client which should be able to save the network cost. **What are the tradeoffs between these two approaches?**

Our Hypothesis: When we bring Engine in-process, it would save the network cost of accessing the WASM runtime, but would break up the data locality as a WASM registry.

First, we will explore some qualitative pros and cons between these two execution architectures. Then, we will look into the actual performance numbers obtained from building an in-process solution and running it under load.

A note on intermediate approaches

In this section we compare two extremes:

1. Running scripts entirely on a separate Shopify service and serviced over a TCP interface.
2. Running scripts in the same unicorn process hosting Shopify/shopify.

These are not the only two approaches. There are several other intermediary solutions, such as:

1. Running scripts in a sidecar container.
2. Running scripts in a different process on the same webserver.

For the sake of this doc we will focus on comparing the extremes, but the best long term architecture might actually sit somewhere in the middle.

In-Process Approach

We explore qualitative pros and cons with the approach of bundling a Wasm runtime with a Ruby gem and running directly in-process over FFI.

Pros

- Run metric is better, as there is (almost) no additional communication cost for WASM execution.
- Less infrastructure risk since we don't have to run Wasm on an external service. There is no remote service to act as a separate point of failure, no separate deployments to manage, etc.

- Don’t have to scale a separate set of infrastructure independently. The solution just scales with Core, assuming we provision enough resources per worker.
- As we expand our data access APIs, having the commerce data available in the same process may make it an easier problem to solve.

Cons

- It’s hard to maintain data locality, as it shares the same lifecycle with Core pods, (~20 mins lifetime during daytime) and requests are round-robin dispatched to web workers. As a result, it’s impossible to maintain an in-memory LRU cache of Wasm binaries, which is particularly bad for flash sales. Compiled code must be stored on Memcached or in a Database.
- If compiled code must be stored in memcached or in a database, and script-service remains the source of truth for these modules, then for global services we must create bespoke data replication schemes to cache modules, OR the source of truth would have to be moved to core, which would limit scripts usage outside of a shop.
- If Wasm binaries exceed 2 MB (this is unlikely), we will not be able to store the binary in MemcacheD and likely need to use an even higher latency service to fetch binaries (such as GCS or MySQL).
- There is a deployment cost. Every change made to Engine has to be released as a gem and deployed through the Core shipping pipeline.
- Running untrusted machine code on Core infrastructure presents a higher incident blast radius. Both in terms of security as well as stability and performance (ex. segfaults, threadpool exhaustion, memory issues and so on, can all easily crash a unicorn worker). Graceful degradation of service is not easily guaranteed. We weigh this more heavily because the tech stack is so new (WebAssembly / young compiler codebases).
- Building new capabilities into the Scripts platform (See the Appendix) will be constrained by the host environment, in that any of these capabilities would only be able to rely on infrastructure APIs injected by the host (Memcached, Core’s MySQL). This infrastructure may be different amongst different hosts.
- Additional memory needs to be provisioned on all core web workers to handle the in-process Wasm VM.

Out-Of-Process

We explore qualitative pros and cons with the approach of running Wasm modules on a dedicated service, outside core, and calling those modules over the network.

Pros

- We maintain a high data locality because we can store binaries and data in a memory LRU in-process where they will be executed.

- CPU and memory resources can be scaled more efficiently, instead of having to apply the incremental resources needed to every single core pod.
- As an independent service, the interaction comes in at run time, there is no deployment coupling. Deploying a change to engine won't need to redeploy Core.
- As an independent service, security becomes a separate concern as no direct resource sharing. Currently, runtime-engine is able to run with a stricter set of Kubernetes security settings using gVisor.
- Building new capabilities into the Scripts platform will be easier since the environment is unconstrained, purpose built to run untrusted Wasm modules, and subject to less risk of breaking critical things (thus enabling more agility).
- Easier to build new clients that call Scripts from other services or apps besides core, since we would be using a TCP interface instead of having to possibly rebuild an in-process Wasm VM.
- Graceful service degradation is feasible. In the worst case of a total outage, the merchant impact is that their scripts don't run. Core will otherwise stay online.

Cons

- There is an unavoidable network cost. Currently, the number is around 1.78-2.6ms single trip from Core to Engine (tier3).
- Preserving fast network time would require a multi-regional deployment, to ensure that Engine is available and located closely to core pods (and storefront renderer pods) around the world.
- At scale, an out-of-process implementation would have to be supported and managed as a global tier 1 service. Graceful degradation is possible, though the merchant impact of an outage could still be very high.

Experiment

To help us better understand the problem, we implemented a prototype to run Engine in process, which we call the in-process-engine. It serves purely as a WASM runtime, and can be deployed as a Gem into core.

To fetch the precompiled WASM module, we chose Memcached as the WASM registry for this test. We put this into production and ran a Genghis with the same criteria as we did for getting the out-of-process baseline performance: 10 mins sample discount script running at discount EP. Here is the performance we get, comparing in-process (source) to out-of-process (source).

Note that we only compare **step 3: Run,** because all other steps are equivalent with both architectures.

Results

	p50	p95
3. Run (in-process)	4.0	11.13
a. fetch (MemcacheD: 26K)	1.12	1.80
b. execute	2.88	9.33
3. Run (out-of-process)	6.49	8.60
a. network	3.56	5.2
b. execute (runtime-engine)	2.93	3.4

Observations

- In-process and out-of-process **execute** time is equivalent, unsurprisingly. We believe the 9.33 ms p95 that we saw with in-process is the result of Ruby GC confounding the data.
- In-process pays the cost of an extra MemcacheD fetch, whereas out-of-process pays the cost of accessing the remote service over the network.

The numbers for in-process are likely close to a **lower bounds,** in terms of what is theoretically possible. This is because the in-process Engine used is a simplified version and assumes that modules are small and can be fetched from MemcacheD. In other words, the execution time reported here for in-process likely won't get too much better in the real world.

Conversely, the numbers for out-of-process represent something closer to an **upper bound**, because there are still further areas of optimization, such as in exploring faster networking protocols and improving the memory LRU.

Lastly, it's worth a reminder that the real performance hurdle (75%+ of the total) here is going to be EP-specific. It will be the code blocks that load data for a script, and handle the result of a script. This could be queries to ActiveRecord or identity cache. It could be expensive implementations of `as_json` across Core. Developers need to do their own component-specific caching, query plan optimization, and other profiling activities to ensure these code blocks are optimized, or else it won't matter how fast the Scripts platform is.

Recommendation

Now that we have qualitative and quantitative observations comparing in-process to out-of-process, we want to make a final recommendation. Let's summarize the key points.

Category	In-Process	Out-of-Process	Winner
Security	Larger blast radius if Scripts gets compromised.	Smaller blast radius and easier to isolate.	Out-of-Process
Core Stability and Uptime	Larger risk to Shopify uptime if there are issues.	Graceful degradation is easier.	Out-of-Process
Module/Data Locality	Limited to MemcacheD (up to 2 MB), GCS or the DB. Must retrieve contents from off-process.	Can guarantee the module is in memory.	Out-of-Process
Infrastructure Complexity	No additional infrastructure. Gem only.	Must maintain dedicated Wasm functions service.	In-Process
Execution Latency	No additional latency; runs right in-line.	Must access service over network.	In-Process
End to End Latency	We pay the cost to fetch the binary from MemcacheD or GCS.	We pay the cost to invoke the script remotely over the network.	Very close.
Time To Market	It will take another few months of development time to bring our prototype into production safely.	Lower; is already in production and stable.	Out-of-Process
Ongoing Costs	-Deploying changes to core is expensive -Must increase resource availability across all core pods. -More expensive to integrate with non-core services. -Building new platform capabilities will be harder due to the constrained environment.	-Ongoing cost to support another critical tier 1 service.	Hard to compare

Surprisingly, execution latency isn't a considerable factor here, at all. The out of process solution is likely better if we wish to optimize purely for performance. The primary downside of an out-of-process solution is the risk and cost of supporting another tier 1 service and deploying it globally. The primary downsides of an in-process solution are that it may slow down or prohibit some of the innovations we hope to build on the scripts platform, and it presents a higher risk to core stability and uptime given the relative immaturity of the tech stack.

For these reasons and some secondary reasons discussed above, we are recommending to continue with an out-of-process architecture.

These key factors are not objective and are full of speculation and judgement. Until we start operating a global Wasm functions service, or until we attempt to build our platform capabilities atop an in-process solution, we won't truly know what the costs will be. Every person who looks at this tradeoff will bring their own historical bias and skillset to this decision.

This is not an irreversible decision. The architecture of our scripts client Gem exposes an abstract "execution" API and as such we can build alternative runtimes and swap them out over time. We can even run multiple runtime architectures concurrently, perhaps split by shop or a beta flag.

There are still some big risks with Scripts v2. Can we create an accessible developer experience that works for partners? Can we create APIs that let Shopifolk easily add extensibility to their components? These are both larger risks than getting the execution architecture wrong, at least for now. Mitigating these risks will require staged rollouts to partners and merchants so we can learn from our users. Throughout these staged rollouts (pre-GA) we will learn a lot more about what it takes to run a global Wasm functions service. If any of these learnings reveal information we did not consider in this document, we will revisit this decision.

Storefront Renderer (SFR) & Alternative Host Environment

How do these recommendations stack up when evaluated through the lens of SFR or any alternate Shopify service from which we wish to invoke scripts?

Runtime Engine Deployment
GCS is the source of truth for Wasm modules, and each runtime engine worker keeps an LRU cache in memory for those modules. Therefore, it’s a service that is easily portable globally:

- The runtime-client gem can route requests to the geographically closest deployment instance of runtime-engine.
- Adding routing rules like Shop affinity in the longer term will allow us to guarantee Wasm modules are almost always hot in the LRU (at the cost of paying for the memory). On the rare occasion they aren’t, we still have a globally distributed backing storage service in GCS.

Ruby Host Lifecycle
In environments where we don’t see the same worker recycling like we do in core, this will only make the ability to cache in memory even easier.

Non-Ruby Services
We would implement the analogous runtime-client library in alternative languages used at Shopify like Go. Since modules are called over a network rather than in-process, it would make the development and support of these clients much easier.

Further Improvements

This section will discuss further improvements we could make for both possible architectures. We also roughly estimate how much improvement we think we could make, for each item. Some of this is touched on in the **what is achievable** section of the baseline report.

In-Process and Out-of-Process

Improvements that are orthogonal to the script runtime architecture and would benefit both runtime architectures.

GraalVM

Shopify's push towards running on TruffleRuby would present some interesting options for Scripts. It could allow us to more easily expand Scripts onto more programming languages, while still providing Wasm compatibility. That said, this is somewhat orthogonal to the in/out-process decision, since an out-of-process solution could also be based on GraalVM and could provide similar benefits. We will closely follow the progress towards running core on TruffleRuby and will happily prototype scripts runtimes when possible.

Argument Serialization

To change the data format from JSON to a more efficient protocol is definitely something we should do. Currently we are still using JSON, which should be replaced with a more compact data format, like MessagePack or ProtoBuf. We didn't make a move on this, as it's considered a less risky decision to make. We should be able to get a performance boost on this.

Input Validation

Currently input is validated on every call, which costs a few ms. We could come up with a heuristics based approach, where input is not validated on every request. Lower level Wasm failures may occur but this could be the right tradeoff in production.

Out-Of-Process

Sharding

We can look at solutions where we route different shop scripts to different service workers. This presents several benefits. We can ensure that any specific bad actor doesn't affect the rest of the merchants, since only a subset of shops would be routed to the same host. This can also let us optimize physical resources, for example ensuring that every script can fit in memory in a subset of the overall worker pool.

Network Optimization

Here we should mainly focus on dragging down the network cost, like using another protocol instead of HTTP/1.1, to achieve the p95 5ms goal. We should be able to achieve comparable latencies to other high performance services used by Shopify such as MemcacheD.

In-Process

Data Locality

If we go with an in-process solution, we should mainly adjust our architecture to make the data (precompiled WASM module) fetching from Client as efficient as possible. We could explore ways to optimize the MemcacheD storage of modules, or invest in alternative options, such as storing the Wasm right with the core deployment.

Appendix: Scripts Platform New Capabilities

Here we summarize a few key capabilities that we see unlocking lots of partner and merchant value in the future when we build them.

Sessions

Suppose scripts were aware of the concept of “session”, such as a “buyer journey” where the buyer moves from the online store through to checkout. If this session token were provided to scripts running, the script could record various events and use those downstream.

Example: Buyer views product X but does not buy it. An upsell script in checkout could see that and render an upsell workflow.

Script Data Storage

Suppose a partner could send shop-specific or app-specific data to load alongside their script. Suppose also that the scripts themselves could read and write to this data store. To stay fast, this data store would have to be colocated with the code itself, ideally hot in memory whenever possible.

Examples: Store list of shipping rates for carriers. Store buyer journey data to make downstream decisions (such as a script that writes to the data store when a buyer views a product page, so that it can be upsold later via another script decision).

Multi Script

Suppose we have extension points where scripts from different apps could be chained or parallelized & joined. In both cases, all relevant modules would need to be available in memory and parallelized across multiple threads.

Example: If a Script can produce a discount, multiple discount producer scripts should be able to run and be joined by some reducer logic.