Google

Introduction to Service Weaver

A Framework for Writing Distributed Applications

https://serviceweaver.dev

Distributed Programming Today

In Our Experience:

Trend

- Split the application into many microservices
- A team owns multiple microservices
- Add new microservices frequently
- Use an internal framework to manage them

Microservices

A piece of code that exports an RPC service.

Distributed Programming Today

In Our Experience:

Trend

- Split the application into many microservices
- A team owns multiple microservices
- Add new microservices frequently
- Use an internal framework to manage them

Why Split

- Scalability, fault tolerance
- Improved agility, maintainability:
  - Multiple languages? But *a vast majority of the teams use only one language.*
  - Different rollout schedules? But *a significant fraction of the teams have only one rollout schedule.*
  - Frequent rollouts? But *only a tiny fraction of the microservices are released very often.*

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Distributed Programming Today

In Our Experience:

Trend

- Split the application into many microservices
- A team owns multiple microservices
- Add new microservices frequently
- Use an internal framework to manage them

Why Split

- Scalability, fault tolerance
- Improved agility, maintainability:
  - Multiple languages? But *a vast majority of the teams use only one language.*
  - Different rollout schedules? But *a significant fraction of the teams have only one rollout schedule.*
  - Frequent rollouts? But *only a tiny fraction of the microservices are released very often.*

But splitting into microservices has drawbacks:

- Versioned upgrades
- Configuration complexity multiplied
- Added IDL and protocol complexity
- API hardening
- E2E and local testing

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today

Monolith

Single binary

Single config

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today

Monolith

Single binary

Single config

Good:

Easier to address many challenges due to using microservices.

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today

Monolith

Single binary

Single config

Good:

Easier to address many challenges due to using microservices.

Bad:

Suffers from challenges that microservices can handle.

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today

Monolith

Single binary

Single config

Good:

Easier to address many challenges due to using microservices.

Bad:

Suffers from challenges that microservices can handle.

Service Weaver

Bridges the gap between the two:

- Programming model of a modular binary
- Flexibility of microservices

Microservices

A piece of code that exports an RPC service.

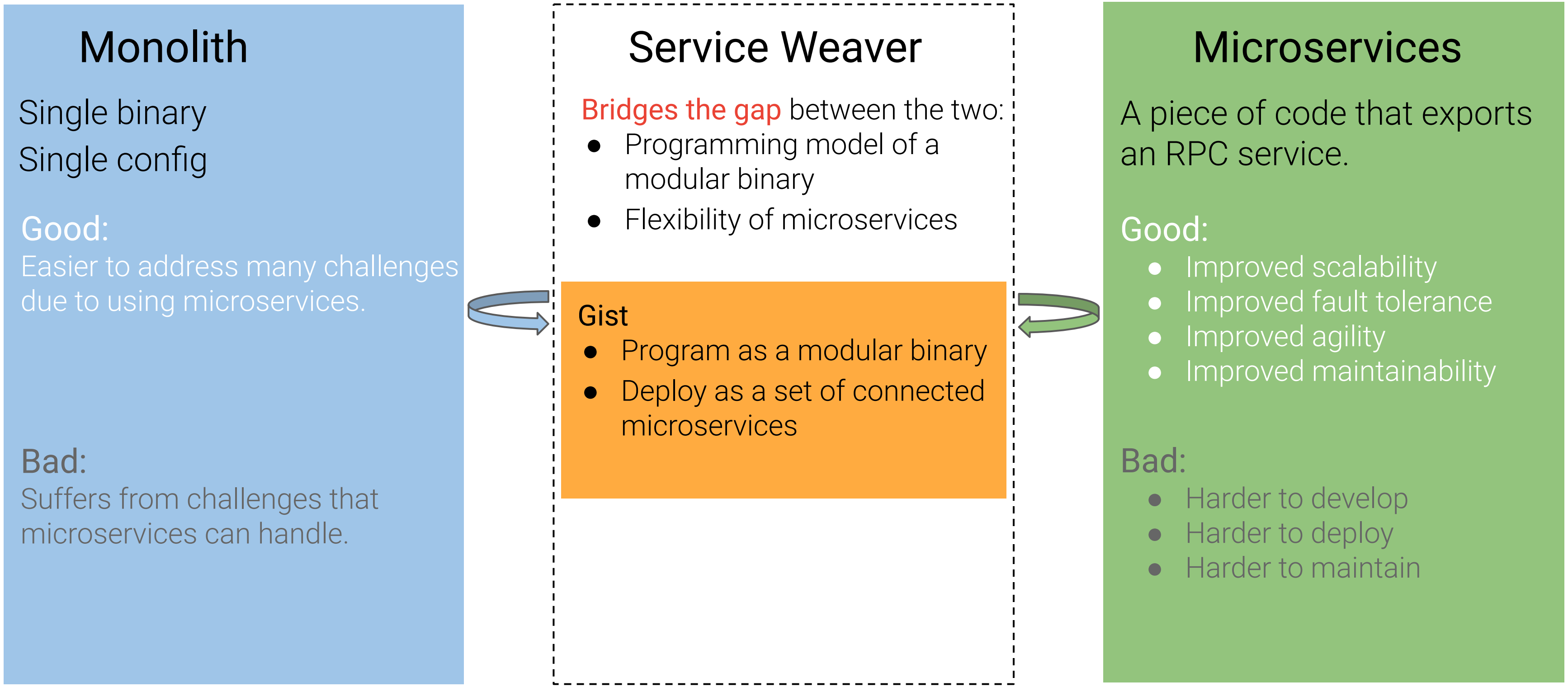
Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today



Monolith

Single binary

Single config

Good:
Easier to address many challenges due to using microservices.

Bad:
Suffers from challenges that microservices can handle.

Service Weaver

Bridges the gap between the two:

- Programming model of a modular binary
- Flexibility of microservices

Gist

- Program as a modular binary
- Deploy as a set of connected microservices

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Distributed Programming Today

Monolith

Single binary
Single config

Good:
Easier to address many challenges due to using microservices.

Bad:
Suffers from challenges that microservices can handle.

Service Weaver

Bridges the gap between the two:

- Programming model of a modular binary
- Flexibility of microservices

Gist

- Program as a modular binary
- Deploy as a set of connected microservices

Enables high-performance applications

Enables portability (multi-cloud, multi-language)

Microservices

A piece of code that exports an RPC service.

Good:

- Improved scalability
- Improved fault tolerance
- Improved agility
- Improved maintainability

Bad:

- Harder to develop
- Harder to deploy
- Harder to maintain

Service Weaver at a Glance

Service Weaver at a Glance

Development

- Using native language constructs
- Organized around native language interfaces
- No code versioning concerns
- Embedded fields to weavify the app

Service Weaver at a Glance

Development

- Using native language constructs
- Organized around native language interfaces
- No code versioning concerns
- Embedded fields to weavify the app

Deployment

- Single binary and a tiny config
- Run as a set of microservices at the same code version
- Multiple deployers (local, GKE, SSH)
- Safe rollouts (blue/green deployments)

Service Weaver at a Glance

Development

- Using native language constructs
- Organized around native language interfaces
- No code versioning concerns
- Embedded fields to weavify the app

Deployment

- Single binary and a tiny config
- Run as a set of microservices at the same code version
- Multiple deployers (local, GKE, SSH)
- Safe rollouts (blue/green deployments)

Telemetry and Testing

- Integrated logging, metrics, and tracing
- Easy local testing
- Quick local iteration over application changes via `go run`

Service Weaver at a Glance

Development

- Using native language constructs
- Organized around native language interfaces
- No code versioning concerns
- Embedded fields to weavify the app

Deployment

- Single binary and a tiny config
- Run as a set of microservices at the same code version
- Multiple deployers (local, GKE, SSH)
- Safe rollouts (blue/green deployments)

Telemetry and Testing

- Integrated logging, metrics, and tracing
- Easy local testing
- Quick local iteration over application changes via `go run`

Performance

- Efficient serialization and transport
- Colocation
- Routing

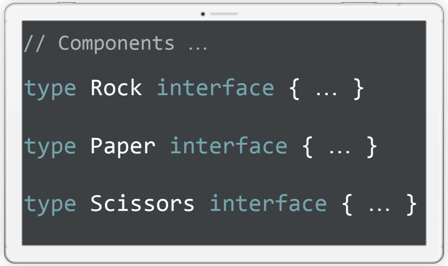
Development

Application

A set of `components` that call each other.

Under the hood: a code generator to weavify the application (e.g., generate encoding, stubs, etc.)

Write as a modular binary



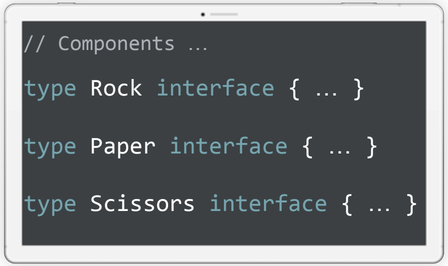
```
// Components ...
type Rock interface { … }
type Paper interface { … }
type Scissors interface { … }
```

Application

A set of `components` that call each other.

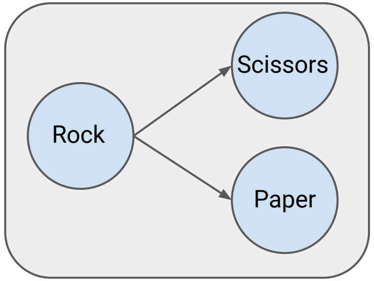
Under the hood: a code generator to weavify the application (e.g., generate encoding, stubs, etc.)

Write as a modular binary



```
// Components ...
type Rock interface { … }
type Paper interface { … }
type Scissors interface { … }
```

Run Locally

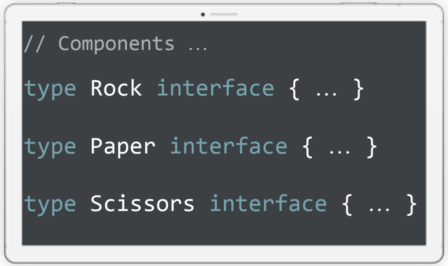


Application

A set of `components` that call each other.

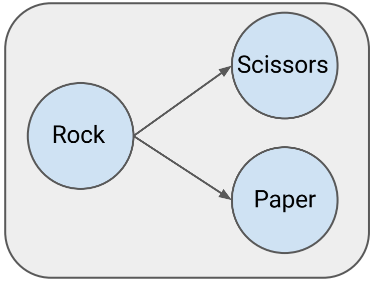
Under the hood: a code generator to weavify the application (e.g., generate encoding, stubs, etc.)

Write as a modular binary

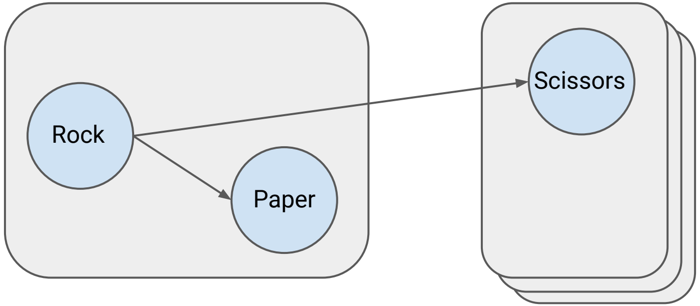


```
// Components ...
type Rock interface { ... }
type Paper interface { ... }
type Scissors interface { ... }
```

Run Locally



Run Distributed



Application

How to define a `component`?

- Represented as a Go interface
- Args/results must be serializable

```
// Cache component definition.
type Cache interface {
  Put(ctx context.Context, key, value string) error
  ...
}
```

Application

How to define a `component`?

- Represented as a Go interface
- Args/results must be serializable

How to implement a `component`?

- As a Go struct
- The implementation should embed `weaver.Implements[T]`

```go
// Cache component definition.
type Cache interface {
  Put(ctx context.Context, key, value string) error
  ...
}



// Cache component implementation.
type cache struct {
  weaver.Implements[Cache] // to weavify the component
  data map[string]string
}

func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  return nil
}
```

Application

How to define a `component`?

- Represented as a Go interface
- Args/results must be serializable

How to implement a `component`?

- As a Go struct
- The implementation should embed `weaver.Implements[T]`

How to instantiate a `component`?

- `weaver.Init`(...) to initialize the application
- `weaver.Get[T]`returns a handle to the `component`

```go
// Cache component definition.
type Cache interface {
  Put(ctx context.Context, key, value string) error
  ...
}



// Cache component implementation.
type cache struct {
  weaver.Implements[Cache] // to weavify the component
  data map[string]string
}

func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  return nil
}



func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
}
```

Application

How to define a `component`?

- Represented as a Go interface
- Args/results must be serializable

How to implement a `component`?

- As a Go struct
- The implementation should embed `weaver.Implements[T]`

How to instantiate a `component`?

- `weaver.Init`(…) to initialize the application
- `weaver.Get[T]` returns a handle to the `component`

How to interact with a `component`?

- Simple method calls

```
// Cache component definition.
type Cache interface {
  Put(ctx context.Context, key, value string) error
  ...
}



// Cache component implementation.
type cache struct {
  weaver.Implements[Cache] // to weavify the component
  data map[string]string
}

func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  return nil
}



func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

Deployment

How to deploy?

Release a single binary

Single Config

- Tiny

```
// Rock Paper Scissors app config.
[serviceweaver]
binary = "./game"
colocate = [  // optional
  ["Rock", "Paper"], ["Scissors"]
]
rollout = "1m" // optional
```

How to deploy?

Release a single binary

Single Config

- Tiny

```
// Rock Paper Scissors app config.
[serviceweaver]
binary = "./game"
colocate = [  // optional
  ["Rock", "Paper"], ["Scissors"]
]
rollout = "1m" // optional
```

```
$ go run .                             # Run in a single process.
```

How to deploy?

Release a single binary

Single Config

- Tiny

```
// Rock Paper Scissors app config.
[serviceweaver]
binary = "./game"
colocate = [  // optional
  ["Rock", "Paper"], ["Scissors"]
]
rollout = "1m" // optional
```

```
$ go run .                         # Run in a single process.
$ weaver multi deploy weaver.toml  # Run in multiple processes.
```

How to deploy?

Release a single binary

Single Config

- Tiny
- Per deployment

Deployment commands for

- Local
- Multiple machines
- Cloud

```
// Rock Paper Scissors app config.
[serviceweaver]
binary = "./game"
colocate = [  // optional
  ["Rock", "Paper"], ["Scissors"]
]
rollout = "1m" // optional


// Deployments config.
[ssh]
locations_file = "./ssh_locations.txt"
```

```
$ go run .                           # Run in a single process.
$ weaver multi deploy weaver.toml    # Run in multiple processes.
$ weaver ssh deploy weaver.toml      # Run in the cluster.
```

How to deploy?

Release a single binary

Single Config

- Tiny
- Per deployment

Deployment commands for

- Local
- Multiple machines
- Cloud

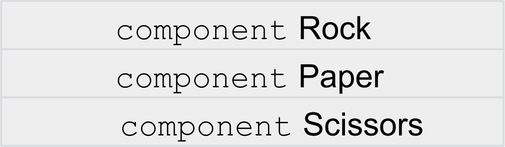
```
// Rock Paper Scissors app config.
[serviceweaver]
binary = "./game"
colocate = [  // optional
  ["Rock", "Paper"], ["Scissors"]
]
rollout = "1m" // optional


// Deployments config.
[ssh]
locations_file = "./ssh_locations.txt"

[gke]
regions = ["us-west1"]
public_listener = [
  {name = "game", hostname = "game.example.com"},
]
```

```
$ go run .                             # Run in a single process.
$ weaver multi deploy weaver.toml      # Run in multiple processes.
$ weaver ssh deploy weaver.toml        # Run in the cluster.
$ weaver gke deploy weaver.toml        # Run in the cloud.
```

How is it deployed?

How is it deployed?

Modular binary



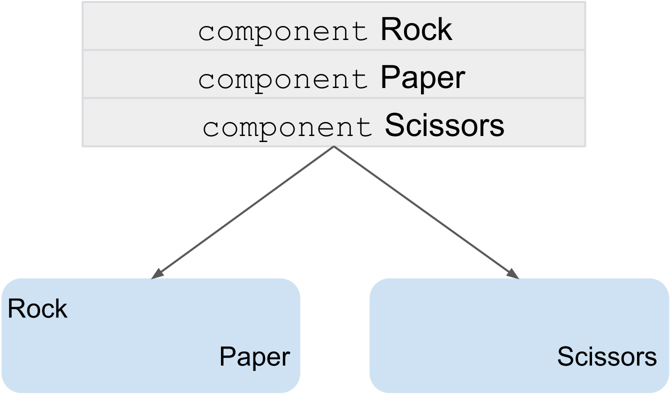
`component` Rock

`component` Paper

`component` Scissors

How is it deployed?

Modular binary



`component` Rock

`component` Paper

`component` Scissors

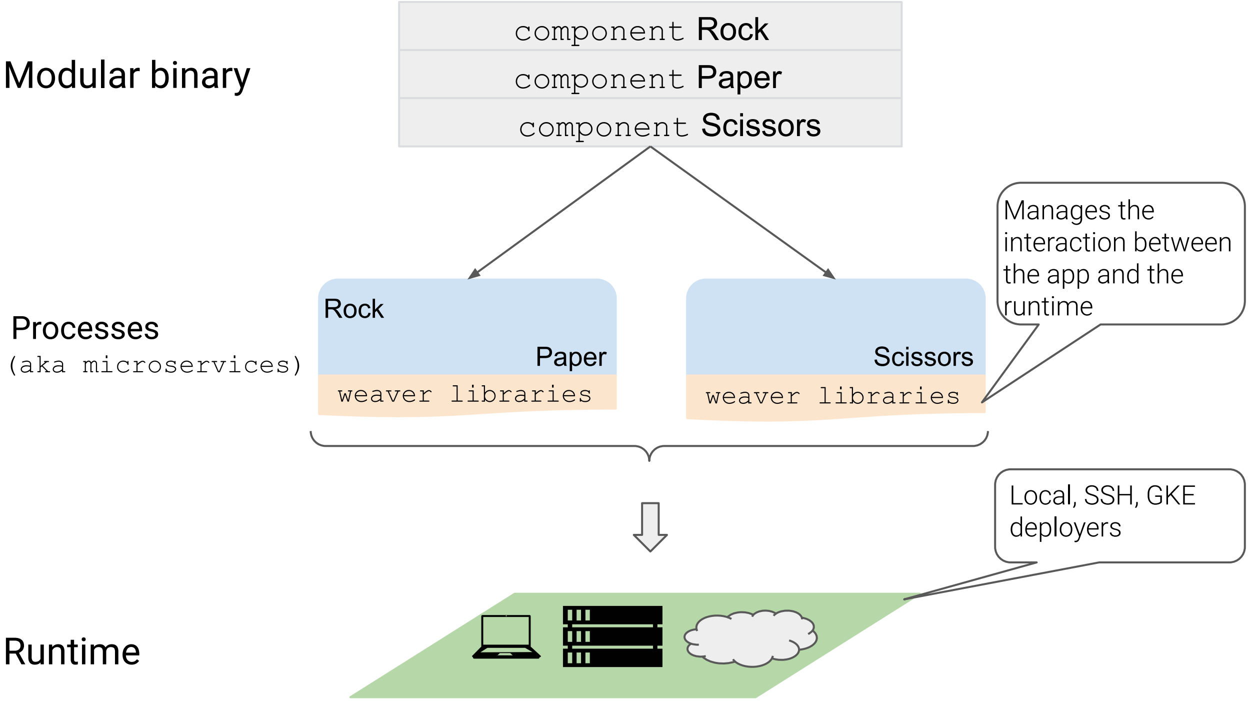
Processes
`(aka microservices)`

Rock

Paper

Scissors

How is it deployed?



Modular binary

`component` Rock

`component` Paper

`component` Scissors

Processes
`(aka microservices)`

Rock

Paper

`weaver libraries`

Scissors

`weaver libraries`

Manages the interaction between the app and the runtime

Runtime

Local, SSH, GKE deployers

Telemetry and Testing

Instrumentation

Logging

- Each `component` has an associated logger
- Structured logging: cat, tail, search, filter logs

```
...
func (c *cache) Put(_ context.Context, key, value string) error {
  c.Logger().Info(“Add”, “key”, key, “value”, value)
  c.data[key] = value
  return nil
}
```

```
$ weaver gke logs --follow  # Follow all the logs.
$ weaver gke logs ‘app==“cache” && level==”info” # Only info logs.
$ …
```

Instrumentation

Logging

- Each `component` has an associated logger
- Structured logging: cat, tail, search, filter logs

```
...
func (c *cache) Put(_ context.Context, key, value string) error {
  c.Logger().Info(“Add”, “key”, key, “value”, value)
  c.data[key] = value
  return nil
}
```

```
$ weaver gke logs --follow  # Follow all the logs.
$ weaver gke logs ‘app==“cache” && level==”info” # Only info logs.
$ ...
```

Metrics

- Counters, gauges, and histograms
- Includes framework metrics

```
...
var putCount = weaver.NewCounter(“put_count”, “Number of Put ops.”)
func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  putCount.Add(1.0)
  return nil
}
```

Instrumentation

Logging

- Each `component` has an associated logger
- Structured logging: cat, tail, search, filter logs

```
...
func (c *cache) Put(_ context.Context, key, value string) error {
  c.Logger().Info(“Add”, “key”, key, “value”, value)
  c.data[key] = value
  return nil
}
```

```
$ weaver gke logs --follow  # Follow all the logs.
$ weaver gke logs ‘app==“cache” && level==”info” # Only info logs.
$ …
```

Metrics

- Counters, gauges, and histograms
- Includes framework metrics

```
...
var putCount = weaver.NewCounter(“put_count”, “Number of Put ops.”)
func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  putCount.Add(1.0)
  return nil
}
```

Tracing

- Relies on OpenTelemetry
- Once enabled, all HTTP requests and `component` method calls are automatically traced

```
func main() {
  …
  // Create an otel handler to enable tracing.
  otelHandler := otelhttp.NewHandler(http.DefaultServeMux, “http”)
  http.Serve(lis, otelHandler)
}
```

Instrumentation

Logging

- Each `component` has an associated logger
- Structured logging: cat, tail, search, filter logs

```
...
func (c *cache) Put(_ context.Context, key, value string) error {
  c.Logger().Info(“Add”, “key”, key, “value”, value)
  c.data[key] = value
  return nil
}
```

```
$ weaver gke logs --follow  # Follow all the logs.
$ weaver gke logs ‘app==“cache” && level==”info” # Only info logs.
$ …
```

Tracing

- Relies on OpenTelemetry
- Once enabled, all HTTP requests and `component` method calls are automatically traced

```
func main() {
  …
  // Create an otel handler to enable tracing.
  otelHandler := otelhttp.NewHandler(http.DefaultServeMux, “http”)
  http.Serve(lis, otelHandler)
}
```

Metrics

- Counters, gauges, and histograms
- Includes framework metrics

```
...
var putCount = weaver.NewCounter(“put_count”, “Number of Put ops.”)
func (c *cache) Put(_ context.Context, key, value string) error {
  c.data[key] = value
  putCount.Add(1.0)
  return nil
}
```

Profiling

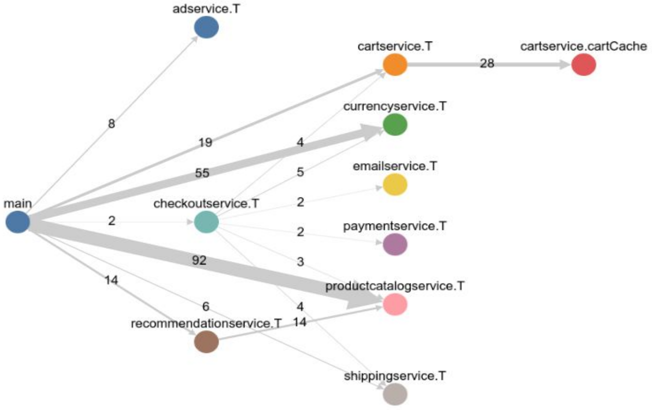
- Profile each individual process and aggregates into a single profile
- Captures the performance of the app as a whole

```
$ weaver gke profile cache             # CPU profile.
$ weaver gke profile --type=heap cache # Heap profile.
$ …
```

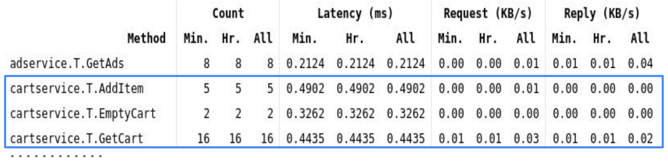
Monitoring

Dashboards

Bird’s eye view



Per component



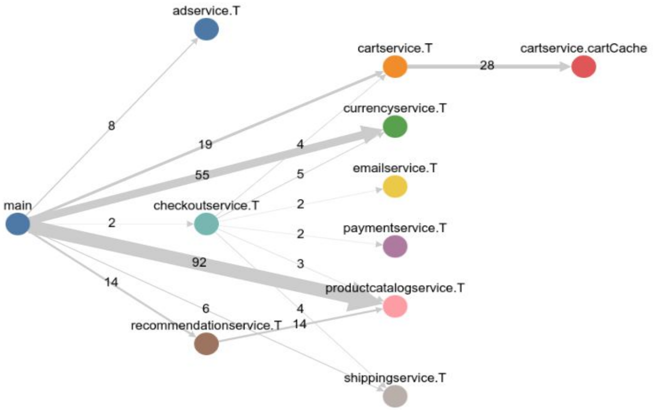
	Count			Latency (ms)			Request (KB/s)			Reply (KB/s)		
Method	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All
adservice.T.GetAds	8	8	8	0.2124	0.2124	0.2124	0.00	0.00	0.01	0.01	0.01	0.04
cartservice.T.AddItem	5	5	5	0.4902	0.4902	0.4902	0.00	0.00	0.01	0.00	0.00	0.00
cartservice.T.EmptyCart	2	2	2	0.3262	0.3262	0.3262	0.00	0.00	0.00	0.00	0.00	0.00
cartservice.T.GetCart	16	16	16	0.4435	0.4435	0.4435	0.01	0.01	0.03	0.01	0.01	0.02

....

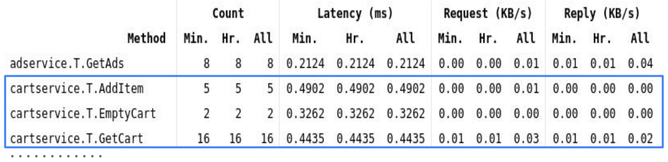
Monitoring

Dashboards

Bird's eye view

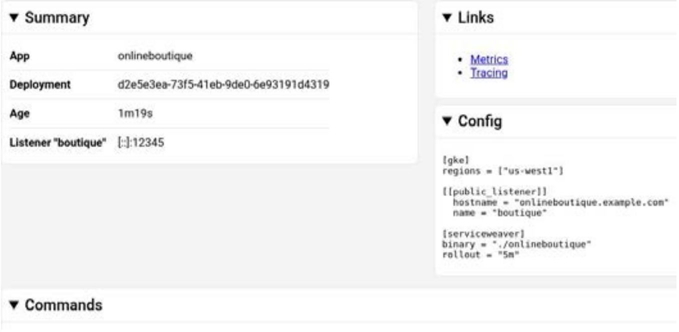


Per component



	Count			Latency (ms)			Request (KB/s)			Reply (KB/s)		
Method	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All
adservice.T.GetAds	8	8	8	0.2124	0.2124	0.2124	0.00	0.00	0.01	0.01	0.01	0.04
cartservice.T.AddItem	5	5	5	0.4902	0.4902	0.4902	0.00	0.00	0.01	0.00	0.00	0.00
cartservice.T.EmptyCart	2	2	2	0.3262	0.3262	0.3262	0.00	0.00	0.00	0.00	0.00	0.00
cartservice.T.GetCart	16	16	16	0.4435	0.4435	0.4435	0.01	0.01	0.03	0.01	0.01	0.02
....												

Integration with Monitoring Frameworks



▼ Summary

App	onlineboutique
Deployment	d2e5e3ea-73f5-41eb-9de0-6e93191d4319
Age	1m19s
Listener "boutique"	[::]:12345

▼ Links

- Metrics
- Tracing

▼ Config

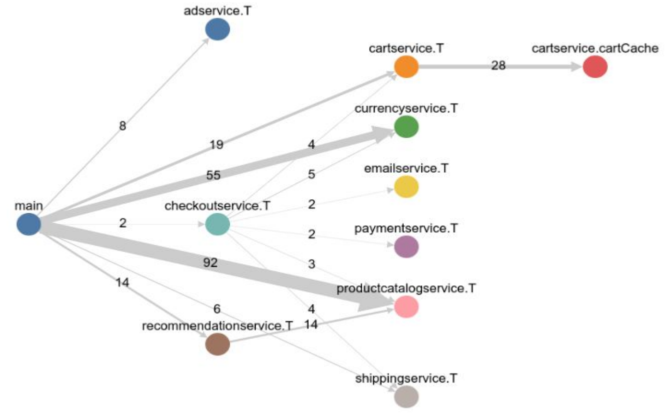
```
[gke]
regions = ["us-west1"]

[[public_listener]]
  hostname = "onlineboutique.example.com"
  name = "boutique"

[serviceweaver]
binary = "./onlineboutique"
rollout = "5m"
```

▼ Commands

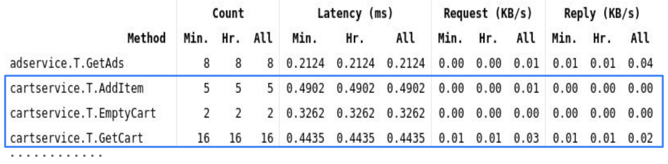
Monitoring

Dashboards

Bird's eye view



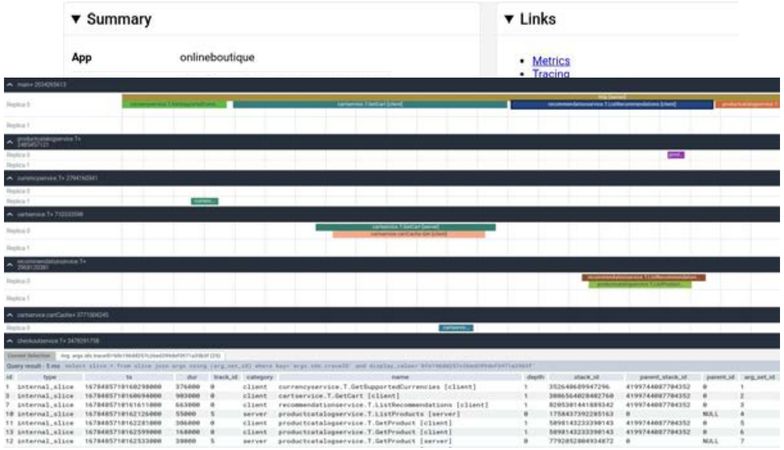
Per component



	Count			Latency (ms)			Request (KB/s)			Reply (KB/s)		
Method	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All
adservice.T.GetAds	8	8	8	0.2124	0.2124	0.2124	0.00	0.00	0.01	0.01	0.01	0.04
cartservice.T.AddItem	5	5	5	0.4902	0.4902	0.4902	0.00	0.00	0.01	0.00	0.00	0.00
cartservice.T.EmptyCart	2	2	2	0.3262	0.3262	0.3262	0.00	0.00	0.00	0.00	0.00	0.00
cartservice.T.GetCart	16	16	16	0.4435	0.4435	0.4435	0.01	0.01	0.03	0.01	0.01	0.02

....

Integration with Monitoring Frameworks



▼ Summary

App onlineboutique

▼ Links

- Metrics
- Tracing

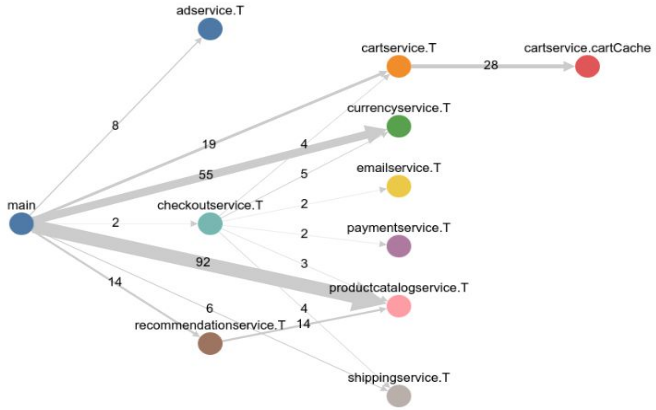
Query result - 5 ms select slice.* from slice join args using (arg_set_id) where key='args.ids.traceID' and display_value='[illegible]'

id	type	ts	dur	track_id	category	name	depth	stack_id	parent_stack_id	parent_id	arg_set_id
1	internal_slice	1678485718160298000	376000	0	client	currencyservice.T.GetSupportedCurrencies [client]	1	3526486089947296	4199744087704352	0	1
3	internal_slice	1678485718160694000	903000	0	client	cartservice.T.GetCart [client]	1	3086564028402768	4199744087704352	0	2
7	internal_slice	1678485718161611000	663000	0	client	recommendationservice.T.ListRecommendations [client]	1	8205301441889342	4199744087704352	0	3
10	internal_slice	1678485718162126000	55000	5	server	productcatalogservice.T.ListProducts [server]	0	1758437392285163	0	NULL	4
11	internal_slice	1678485718162281000	306000	0	client	productcatalogservice.T.GetProduct [client]	1	5098143233390143	4199744087704352	0	5
13	internal_slice	1678485718162599000	168000	0	client	productcatalogservice.T.GetProduct [client]	1	5098143233390143	4199744087704352	0	6
12	internal_slice	1678485718162533000	38000	5	server	productcatalogservice.T.GetProduct [server]	0	7792852004934872	0	NULL	7

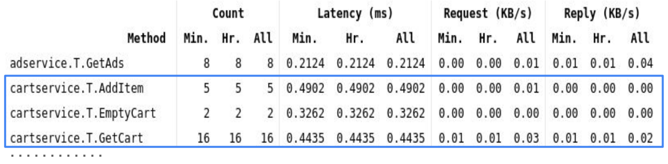
Monitoring

Dashboards

Bird's eye view



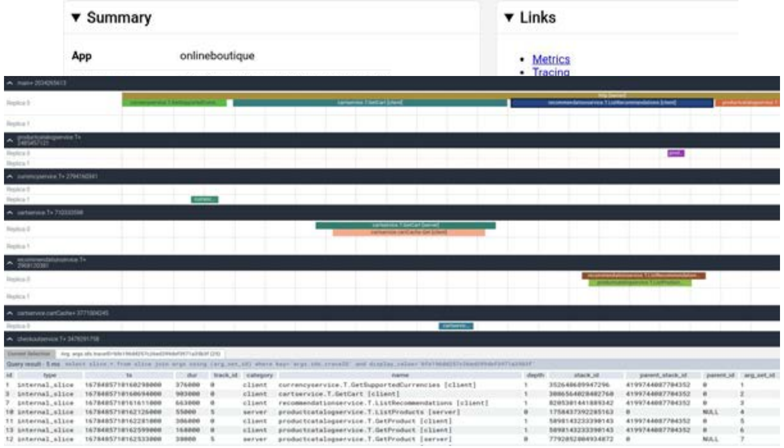
Per component



	Count			Latency (ms)			Request (KB/s)			Reply (KB/s)		
Method	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All
adservice.T.GetAds	8	8	8	0.2124	0.2124	0.2124	0.00	0.00	0.01	0.01	0.01	0.04
cartservice.T.AddItem	5	5	5	0.4902	0.4902	0.4902	0.00	0.00	0.01	0.00	0.00	0.00
cartservice.T.EmptyCart	2	2	2	0.3262	0.3262	0.3262	0.00	0.00	0.00	0.00	0.00	0.00
cartservice.T.GetCart	16	16	16	0.4435	0.4435	0.4435	0.01	0.01	0.03	0.01	0.01	0.02

....

Integration with Monitoring Frameworks



▼ Summary

App onlineboutique

▼ Links

- Metrics
- Tracing

Query result - 5 ms

id	type	ts	dur	track_id	category	name	depth	stack_id	parent_stack_id	parent_id	arg_set_id
1	internal_slice	1678485710160298000	376000	0	client	currencyservice.T.GetSupportedCurrencies [client]	1	3526486089947296	4199744087704352	0	1
3	internal_slice	1678485710160694000	903000	0	client	cartservice.T.GetCart [client]	1	3086564028482760	4199744087704352	0	2
7	internal_slice	1678485710161611000	663000	0	client	recommendationservice.T.ListRecommendations [client]	1	8205301441889342	4199744087704352	0	3
10	internal_slice	1678485710162126000	55000	5	server	productcatalogservice.T.ListProducts [server]	0	1758437392285163	0	NULL	4
11	internal_slice	1678485710162281000	306000	0	client	productcatalogservice.T.GetProduct [client]	1	5898143233390143	4199744087704352	0	5
13	internal_slice	1678485710162599000	168000	0	client	productcatalogservice.T.GetProduct [client]	1	5898143233390143	4199744087704352	0	6
12	internal_slice	1678485710162533000	30000	5	server	productcatalogservice.T.GetProduct [server]	0	7792852004934872	0	NULL	7

Tracing:

- Perfetto
- Google Cloud Trace

Metrics:

- Prometheus
- Metrics Explorer

Logs:

- Logs Explorer

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

```
// TestCache tests the Cache component.
func TestCache(t *testing.T) {
  ctx := context.Background()
  root := weavertest.Init(ctx, t, weavertest.Options{})
  cache, err := weaver.Get[Cache](root)
  err = cache.Put(ctx, “mykey”, “myvalue”)
  got , err := cache.Get(ctx, “mykey”)
  if want := “myvalue”; got != want {
    t.Fatal(“got %q, want %q”, got, want)
  }
}
```

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

⇩

```
// TestCache tests the Cache component.
func TestCache(t *testing.T) {
  ctx := context.Background()
  root := weavertest.Init(ctx, t, weavertest.Options{})
  cache, err := weaver.Get[Cache](root)
  err = cache.Put(ctx, “mykey”, “myvalue”)
  got , err := cache.Get(ctx, “mykey”)
  if want := “myvalue”; got != want {
    t.Fatal(“got %q, want %q”, got, want)
  }
}
```

E2E testing

- Use status commands
- Check logs, metrics, traces, dashboards
- Profiles

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

```
// TestCache tests the Cache component.
func TestCache(t *testing.T) {
  ctx := context.Background()
  root := weavertest.Init(ctx, t, weavertest.Options{})
  cache, err := weaver.Get[Cache](root)
  err = cache.Put(ctx, “mykey”, “myvalue”)
  got , err := cache.Get(ctx, “mykey”)
  if want := “myvalue”; got != want {
    t.Fatal(“got %q, want %q”, got, want)
  }
}
```

E2E testing

- Use status commands
- Check logs, metrics, traces, dashboards
- Profiles

(1)

```
# Run in a single process.
~/cache $ go run .
```

- Test whether the app still runs properly

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```go
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

```go
// TestCache tests the Cache component.
func TestCache(t *testing.T) {
  ctx := context.Background()
  root := weavertest.Init(ctx, t, weavertest.Options{})
  cache, err := weaver.Get[Cache](root)
  err = cache.Put(ctx, “mykey”, “myvalue”)
  got , err := cache.Get(ctx, “mykey”)
  if want := “myvalue”; got != want {
    t.Fatal(“got %q, want %q”, got, want)
  }
}
```

E2E testing

- Use status commands
- Check logs, metrics, traces, dashboards
- Profiles

(1)

```
# Run in a single process.
~/cache $ go run .
```

- Test whether the app still runs properly

(2)

```
# Run in multiple processes.
~/cache $ weaver multi deploy weaver.toml
```

- Test whether the app is making any assumptions that don’t hold in a distributed setting

Testing

Unit testing

- Use `weavertest` package
- Run tests in single/multi process mode

```
func main() {
  ctx := context.Background()
  root := weaver.Init(ctx) // Initialize the app.
  cache, err := weaver.Get[Cache](root)
  …
  err = cache.Put(ctx, “mykey”, “myvalue”)
  …
}
```

```
// TestCache tests the Cache component.
func TestCache(t *testing.T) {
  ctx := context.Background()
  root := weavertest.Init(ctx, t, weavertest.Options{})
  cache, err := weaver.Get[Cache](root)
  err = cache.Put(ctx, “mykey”, “myvalue”)
  got , err := cache.Get(ctx, “mykey”)
  if want := “myvalue”; got != want {
    t.Fatal(“got %q, want %q”, got, want)
  }
}
```

E2E testing

- Use status commands
- Check logs, metrics, traces, dashboards
- Profiles

(1)

```
# Run in a single process.
~/cache $ go run .
```

- Test whether the app still runs properly

(2)

```
# Run in multiple processes.
~/cache $ weaver multi deploy weaver.toml
```

- Test whether the app is making any assumptions that don’t hold in a distributed setting

(3)

```
# Emulate GKE runs.
~/cache $ weaver gke-local deploy weaver.toml
```

- Test whether the app still works in the presence of multiple app versions running

Performance

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing.

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Benchmarking

- OnlineBoutique Application
- 11 microservices
- E2-Medium VMs (1 core each), GKE, us-west1, 670 qps load
- Non-Weaver vs. Weaver (split) vs. Weaver (merged)

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing.

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Benchmarking

- OnlineBoutique Application
- 11 microservices
- E2-Medium VMs (1 core each), GKE, us-west1, 670 qps load
- Non-Weaver vs. Weaver (split) vs. Weaver (merged)

Metric	Non-Weaver	Weaver (split)	Weaver (merged)	Gains
Go code	2647 lines	2117 lines	2117 lines	up to 1.25x

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing.

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Benchmarking

- OnlineBoutique Application
- 11 microservices
- E2-Medium VMs (1 core each), GKE, us-west1, 670 qps load
- Non-Weaver vs. Weaver (split) vs. Weaver (merged)

Metric	Non-Weaver	Weaver (split)	Weaver (merged)	Gains
Go code	2647 lines	2117 lines	2117 lines	up to 1.25x
Config code	1507 lines	9 lines	12 lines	∞

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing.

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Benchmarking

- OnlineBoutique Application
- 11 microservices
- E2-Medium VMs (1 core each), GKE, us-west1, 670 qps load
- Non-Weaver vs. Weaver (split) vs. Weaver (merged)

Metric	Non-Weaver	Weaver (split)	Weaver (merged)	Gains
Go code	2647 lines	2117 lines	2117 lines	up to 1.25x
Config code	1507 lines	9 lines	12 lines	∞
Autoscaled to	21 VMs	10 VMs	5 VMs	up to 4x

Highly-Performant Runtime

Efficient encoding/decoding

- Argument/result types known at the sender/receiver
- No versioning overheads

Efficient transport

- Built on top of TCP
- Custom load-balancing.

Colocation

- Flexibility to colocate some `components` in the same OS process

Routing

- Increased likelihood to route requests with the same key to the same `component` replica
- Increases cache hit ratio

Benchmarking

- OnlineBoutique Application
- 11 microservices
- E2-Medium VMs (1 core each), GKE, us-west1, 670 qps load
- Non-Weaver vs. Weaver (split) vs. Weaver (merged)

Metric	Non-Weaver	Weaver (split)	Weaver (merged)	Gains
Go code	2647 lines	2117 lines	2117 lines	up to 1.25x
Config code	1507 lines	9 lines	12 lines	∞
Autoscaled to	21 VMs	10 VMs	5 VMs	up to 4x
Median latency	40 ms	12 ms	6 ms	up to 7x
99p latency	520 ms	130 ms	14 ms	up to 37x

FAQ

Do’s

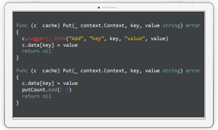
- Write a single modularized binary
- Decide on how to split into microservices only when you deploy
- Don't worry about the underlying network transports (e.g., HTTP, gRPC) and serialization (e.g., JSON, Protocol Buffers)
- Allows cross-component calls within the same process to be optimized down to local method calls

Don’ts

- Hide the network - the method calls should be treated as remote by default
- Organize the application code and low level interactions through an IDL
- Worry about code versioning issues and rollouts

Service Weaver

A Framework for Writing Distributed Applications



```
func (c *cache) Put(_ context.Context, key, value string) error
{
  c.Logger().Info("Add", "key", key, "value", value)
  c.data[key] = value
  return nil
}

func (c *cache) Put(_ context.Context, key, value string) error
{
  c.data[key] = value
  putCount.Add(1.0)
  return nil
}
```

Easy to Develop

Split into `Components`

Interact through method calls

Single binary

Easy to Deploy

Tiny Config

Deployed as microservices

Local, SSH, GKE deployers

Easy to Monitor

Embedded telemetry

Integration with Monitoring Frameworks

High-Performance

Efficient encoding

Efficient transport

Component Colocation

http://serviceweaver.dev

Try it out!

Contribute!

Give Feedback!

Google