DEBUGGING MICROSERVICES

Idit Levine

solo.io

About me

Idit Levine
Founder and CEO of solo.io

@Idit_Levine

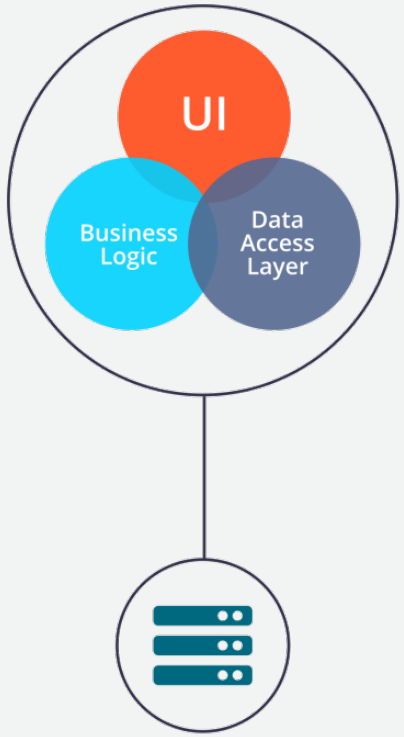
@ilevin

solo.io

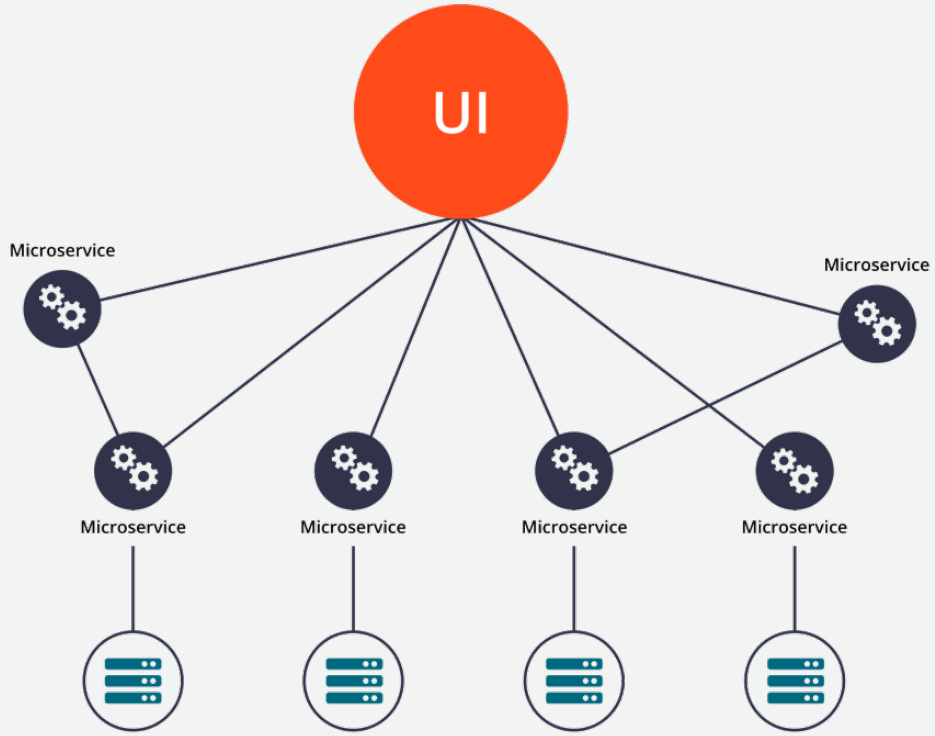
The problem:

Debugging microservices applications is hard

The problem



Monolithic Architecture



Microservice Architecture

A monolithic application consists of **a single process**

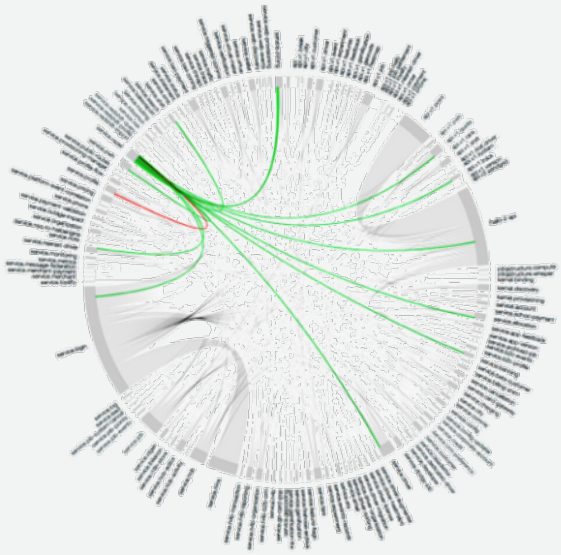
An attached debugger allows viewing the complete state of the application during runtime

A microservices application consists of potentially **hundreds of processes**

Is it possible to get a complete view of the state of a such application?!

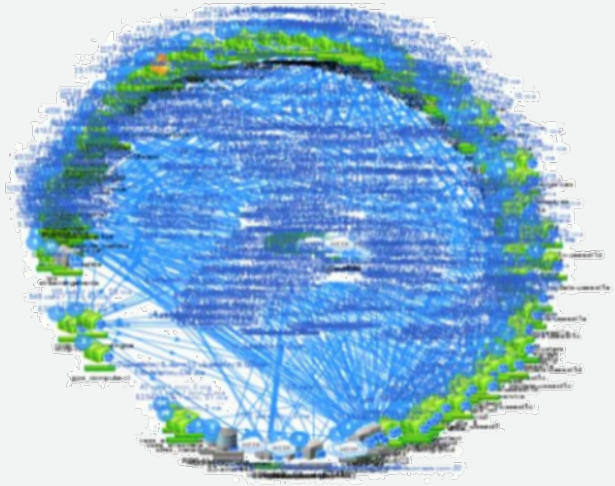
The problem

450+ microservices



HAIL O

500+ microservices



NETFLIX

500+ microservices

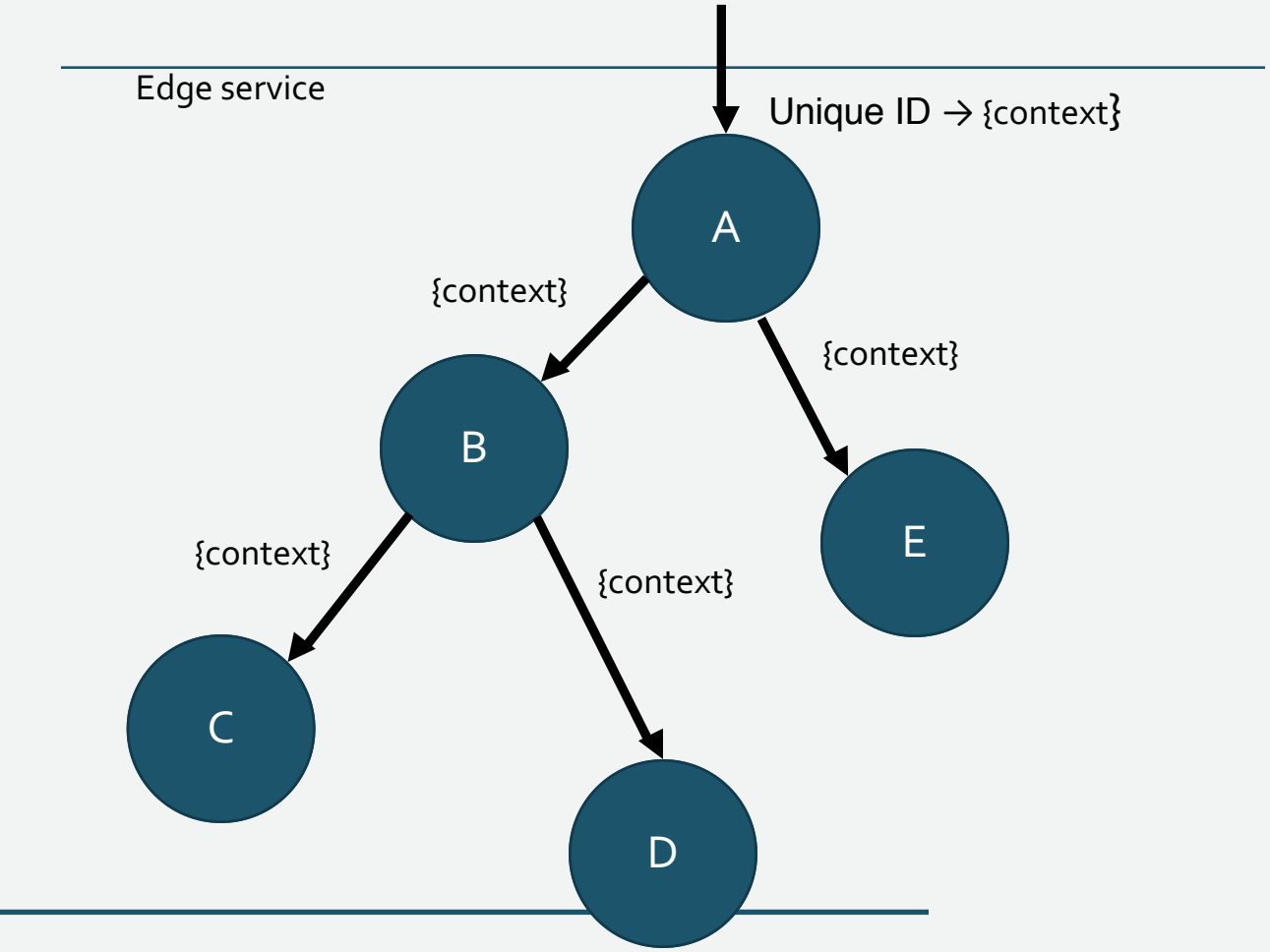
The problem


Honest Status Page
@honest_update
Follow
We replaced our monolith with micro services so that every outage could be more like a murder mystery.
4:10 PM - 7 Oct 2015
3,028 Retweets 2,476 Likes
20
3.0K
2.5K

Solution I

OpenTracing

OpenTracing



1. assign a ***unique identifier*** to each request at the edge service

2. store it in a ***context*** object, along with other metadata

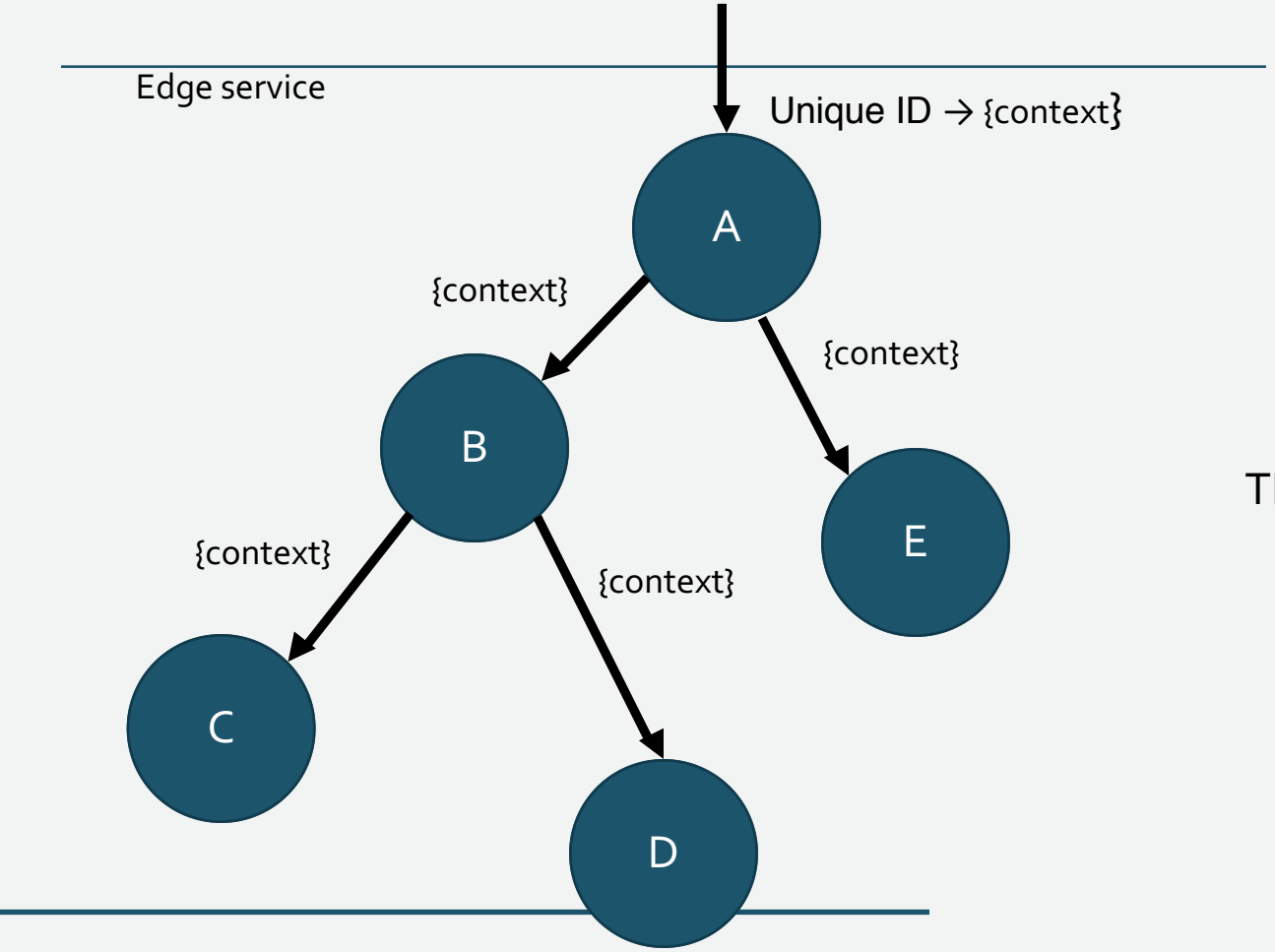
3. ***propagate the context*** across process boundaries (in-band)

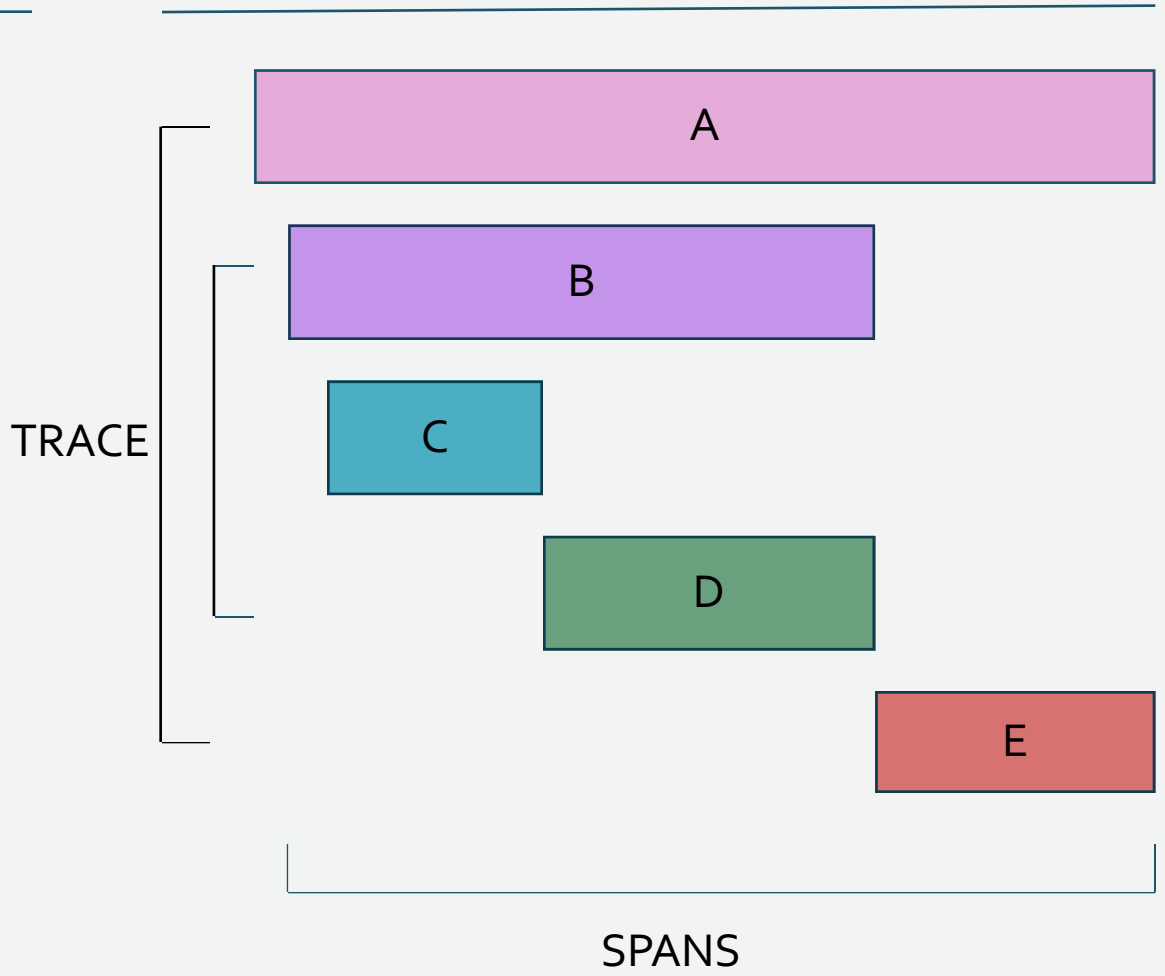
4. ***baggage*** is arbitrary K/V

5. capture timing, events, tags and collect them out of band (async)

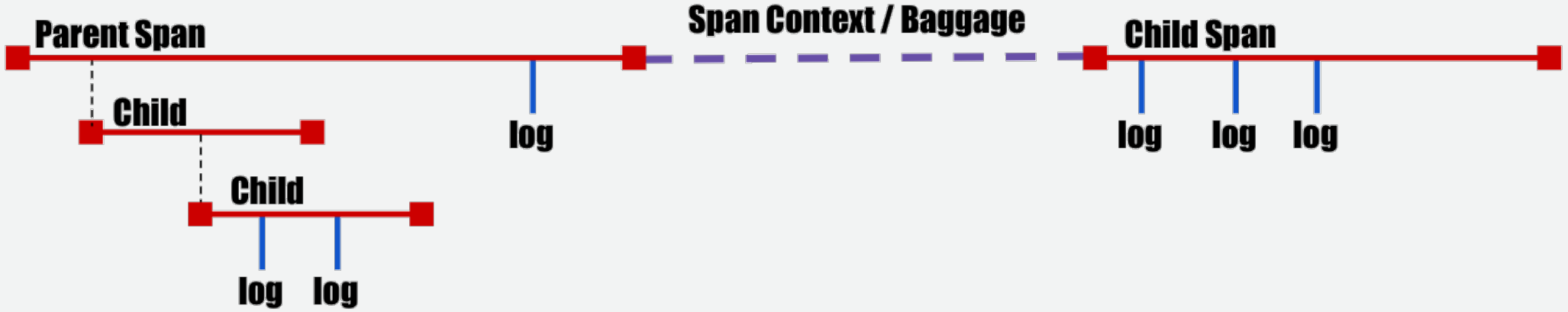
6.re-assemble the call tree from the storage for the UI

OpenTracing





OpenTracing Architecture

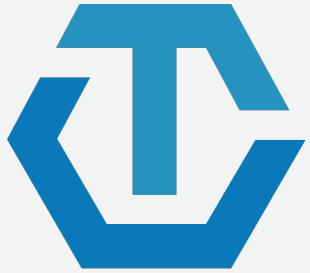


spans - basic unit of timing and causality. can be tagged with key/value pairs. logs - structured data recorded on a span.

span context - serializable format for linking spans across network boundaries. carries baggage, such as a request and client IDs.

tracers - anything that plugs into the OpenTracing API to record information. zipKin, jaeger & lightstep. but also metrics (Prometheus) and logging.

OpenTracing



OpenTracing is a consistent, expressive, vendor-neutral APIs for popular platforms, OpenTracing makes it easy for developers to add (or switch) tracing implementations with an O(1) configuration change.

CLOUD NATIVE
COMPUTING FOUNDATION

OpenTracing Demo

OpenTracing uses

logging - easy to output to any logging tool, even from OSS components.

metrics/alerting - measure based on tags, span timing, log data.

context propagation - use baggage to carry request and user ID’s, etc.

critical path analysis - drill down into request latency in very high fidelity.

system topology analysis - identify bottlenecks due to shared resources.

OpenTracing limitations

openTracing does not provide ***run-time debugging***

openTracing requires ***wrapping and changing the code***

no holistic view of the application state – can ***only see what was printed***

the ***process*** (repeatedly modify the application and test) ***is expansive***

Impossible to change variable values in runtime

logging and printing results in ***performances overhead***

Solution II

Squash

Squash brings the power of modern popular debuggers to developers of microservices apps that run on container orchestration platforms.

Squash bridges between the orchestration platform (without changing it) and IDE.

With Squash, you can:

- Live debugging cross multi microservices
- Debug container in a pod
- Debug a service
- Set breakpoints
- Step through the code
- View and modify values of variables
- and more ...

Squash Demo

Squash Architecture

Squash server: holds the information about the breakpoints for each application, orchestrates and controls the squash clients. Squash server deploys and runs on Kubernetes

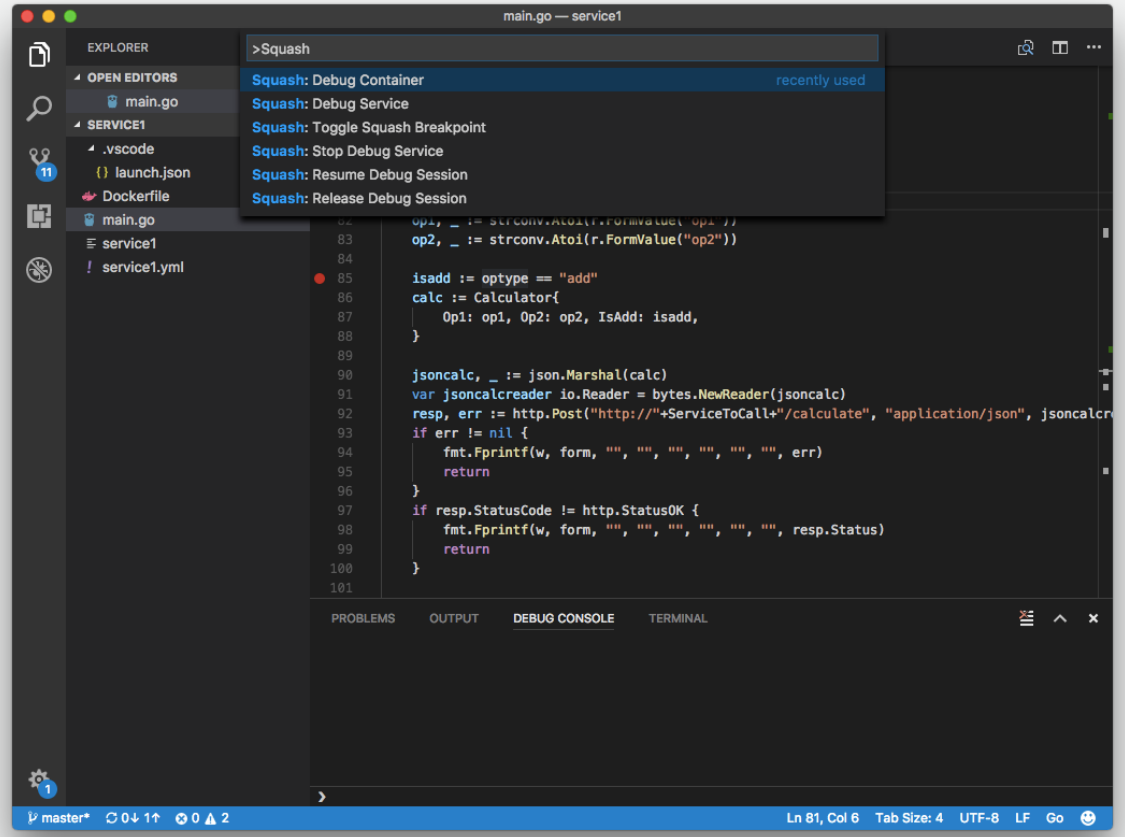
Squash client: deploys as daemon set on Kubernetes node. The client wraps as docker container, and also contains the binary of the debuggers.

Squash User Interface: squash uses IDEs as its user interface. After installing the Squash extension, Squash commands are available in the IDE command palette.

What vegetable scares all the bugs? Squash!"

one of my 8-year old daughter's favorite riddles

Squash Architecture: vs code extension



vs code extension ➜ ***kubectl*** to present the user pod/container/debugger options

vs code extension ➜ ***Squash server*** with debug config (pod/container/debugger /breakpoint) ➜ waits for debug session

vs code extension ➜ connects to the debug server & transfers control to the native debug extension.

Squash Architecture: Squash Server

vs code extension ➜ ***Squash server***

Squash server ➜ relevant ***Squash client*** with debug config (pod/container/debugger /breakpoint)

Squash server ➜ waits for debug session

Squash Architecture: Squash Client

Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

Squash high level Architecture

Platforms

IDEs

Debuggers

Squash high level Architecture

Platforms

IDEs

Debuggers

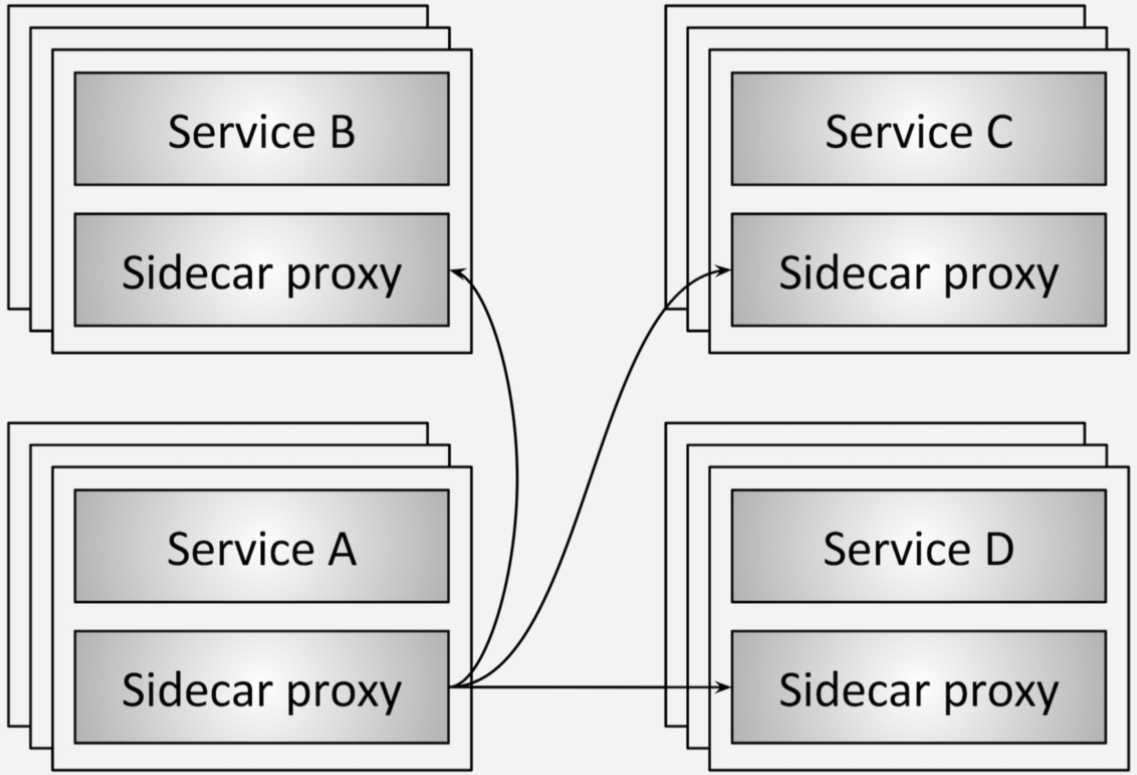
Solution III

Service mesh

Service Mesh

Service mesh data plane:
Touches every packet/request in the system. Responsible for ***service discovery***, ***health checking***, ***routing***, ***load balancing***, ***authentication/authorization***, and ***observability***.

Service mesh control plane:
Provides ***policy*** and ***configuration*** for all the running data planes in the mesh. Does not touch any packets/requests in the system. ***The control plane turns all of the data planes into a distributed system***.



Envoy – data plane

Out of process architecture: developers to focus on business logic

Modern C++11 code base: Fast and productive.

L3/L4 filter architecture: Can be used for things other than HTTP (TCP proxy at its core)

HTTP L7 filter architecture: Make it easy to plug in different functionality.

HTTP/2 first! (Including gRPC and a nifty gRPC HTTP/1.1 bridge).

Service discovery and **active health checking.**

Advanced ***load balancing***: Retry, timeouts, circuit breaking, rate limiting, shadowing, etc.

Best in class ***observability***: stats, logging, and tracing.

Edge proxy: ***routing*** and ***TLS***.

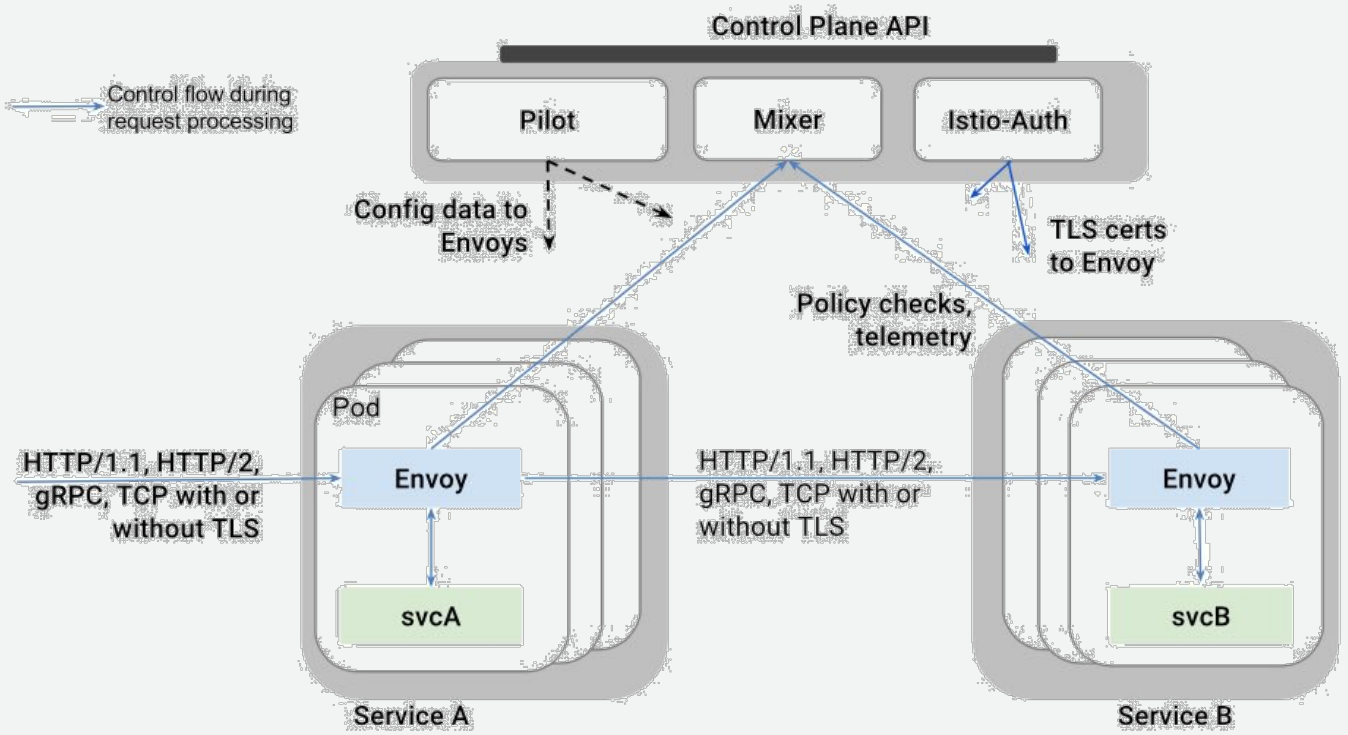
The network should be transparent to applications. When network and application problems do occur it should be easy to determine the source of the problem.

Istio – control plane

Pilot: responsible for the lifecycle of Envoy instances deployed across the Istio service mesh. Pilot exposes APIs for ***service discovery***, dynamic updates to ***load balancing pools*** and ***routing tables***.

Mixer: provides ***Precondition Checking*** (authentication, ACL checks and more), ***Quota Management*** and ***Telemetry Reporting.***

Istio-Auth enhance the security of microservices and their communication without requiring service code changes.



Istio Architecture

Towards an integrated solution

Service mesh, OpenTracing, and Squash

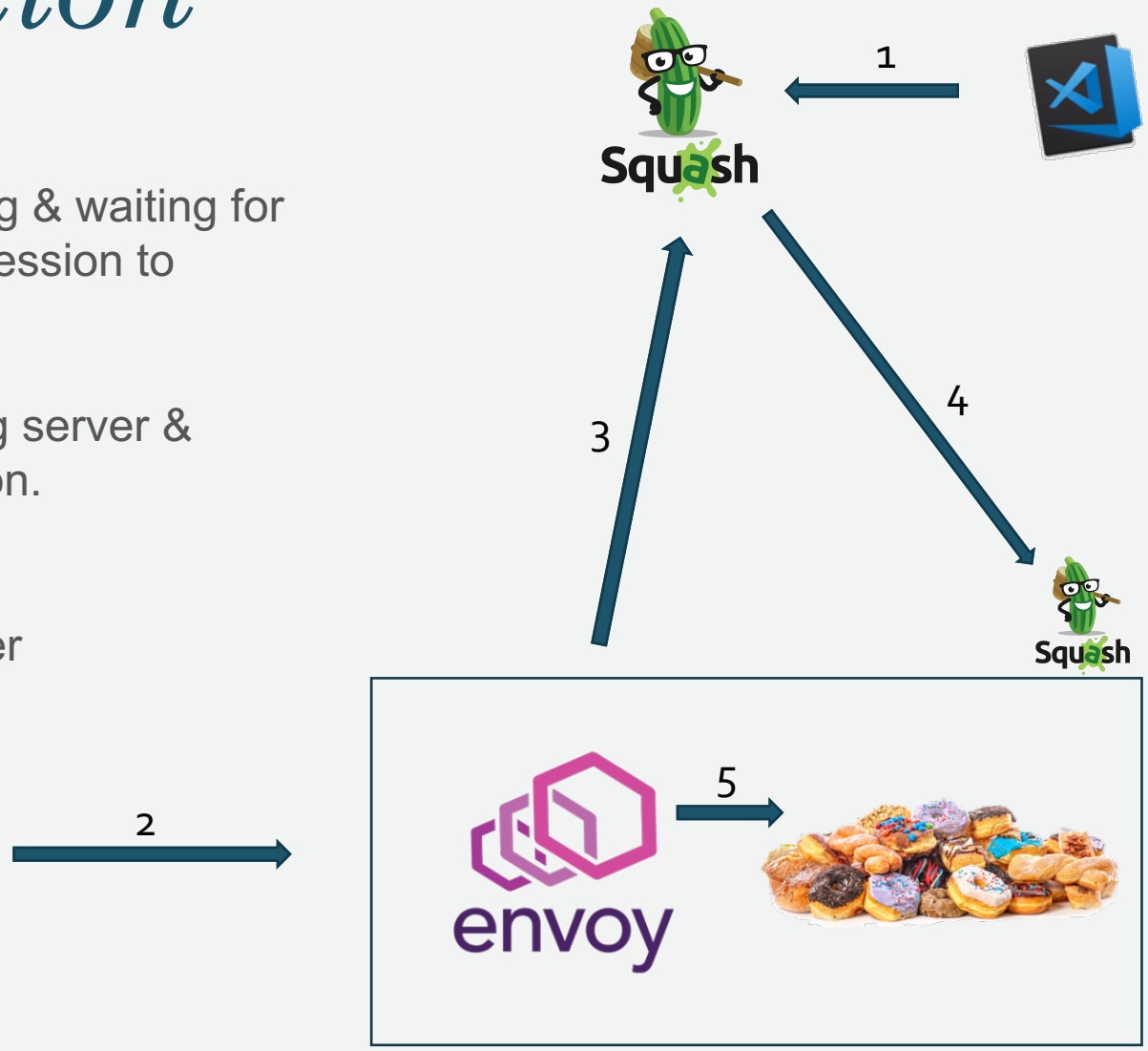
The whole solution

Step 1:
vs code extension ➜ ***Squash server*** pulling & waiting for debug config (service & image) with debug session to connect.

vs code extension ➜ connects to the debug server & transfers control to the native debug extension.

Step 2:
envoy gets a curl request with squash header

Step 3:
➜ ***envoy*** initiate debug config
➜ ***Squash server***
➜ ***envoy*** wait for debug session.



The whole solution

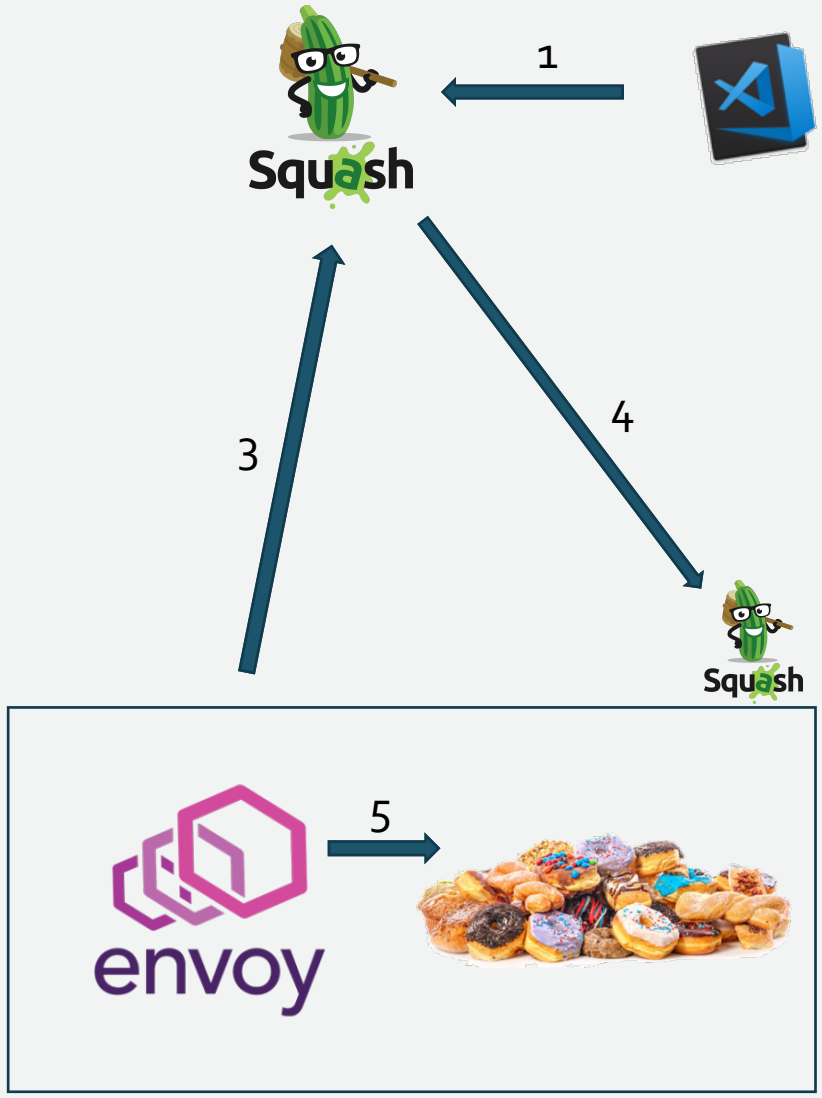
Step 4:
Squash server ➜ ***Squash client***

Squash client ➜ ***container runtime interface*** (to obtain the container host pid)

Squash client ➜ runs the debugger, attaches it to the process in the container, and sets the application breakpoints

Squash client ➜ return debug session.

Step 5:
envoy resume traffic to the app



Service Mesh Demo