User Journey Troubleshooting Tool

Status: Final
Authors: Chuan Chen
Repo: https://github.com/googleinterns/userjourneytool
Last Updated: 2020-09-11

Objective

Many modern software projects consist of an extensive set of interconnected services. Each user journey through the application relies on a subset of such services to function properly. As such, when issues related to a specific user journey arise, it can often be difficult to quickly identify the underlying source of the issue, in order to take further remedial action.

We aim to reduce debugging time by enabling engineers to quickly identify the root cause of issues, given a description (ticket, bug report) specifying the affected user journeys. Moreover, we attempt to reduce communication overhead by providing a centralized location to view system status -- avoiding the need to ask around many people or for a single engineer to be asked repeatedly to relay an issue. As an auxiliary objective, we want to create a form of living documentation describing the relationship between various services, and provide a more detailed view to supplement the simplified, high-level diagrams from manually generated documentation.

We propose a User Journey Troubleshooting Tool to provide engineers with a graphical representation of the service dependency topology, along with the status, SLO/SLI (service-level objectives/indicators), and owner information of each service, as they relate to each user journey.

Goals

- Reduction of debugging time for engineers, especially those on-duty/on-call
- Reduction of communication overhead during the debugging and development process
- Provide a comprehensive, understandable view of system topology and the status of each component and user journey
- Communicate the relationship between user journeys and system components, and their respective statuses

Non-goals

- Providing a visualization of live network traffic between services
- Determining the status/SLI of a service. This is delegated to the service itself.
    - Defining a service's dependencies and SLO targets
- Providing a tool to manually draw a dependency graph at runtime

Background

Various open source tools exist for similar applications.

1. Haystack from Expedia, in particular, the Service Graph Subsystem
    a. Haystack is a server that addresses a similar problem domain, to “facilitate detection and remediation of problems in microservices and websites.” To use Haystack, client services write OpenTracing API *spans* (a unit of tracing data), to a Haystack Agent. The service graph subsystem uses these spans to generate a dependency graph visualization. The service graph displays service health state, traffic information, and interactions in a nice UI.

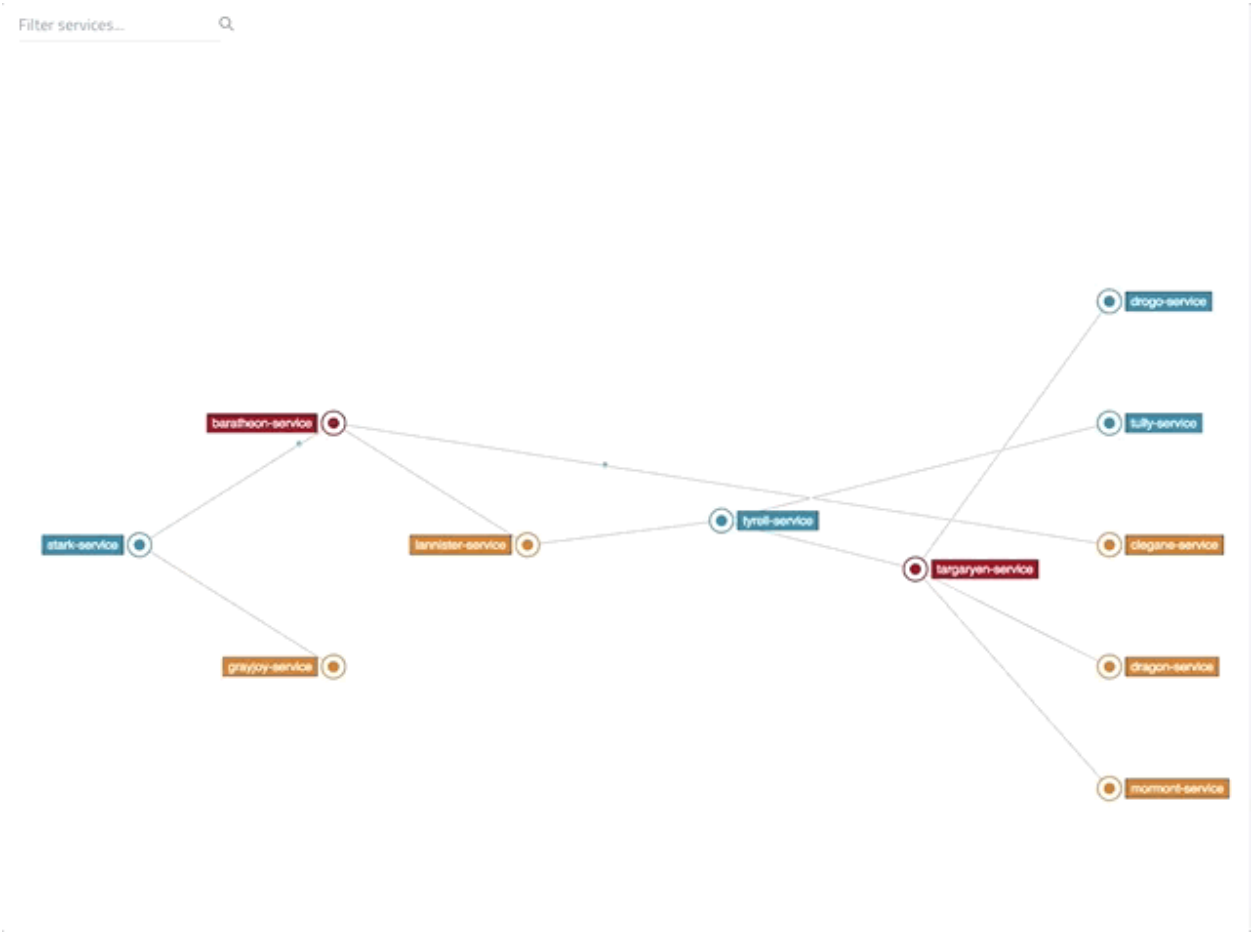

Fig. 1: Haystack Service Graph UI

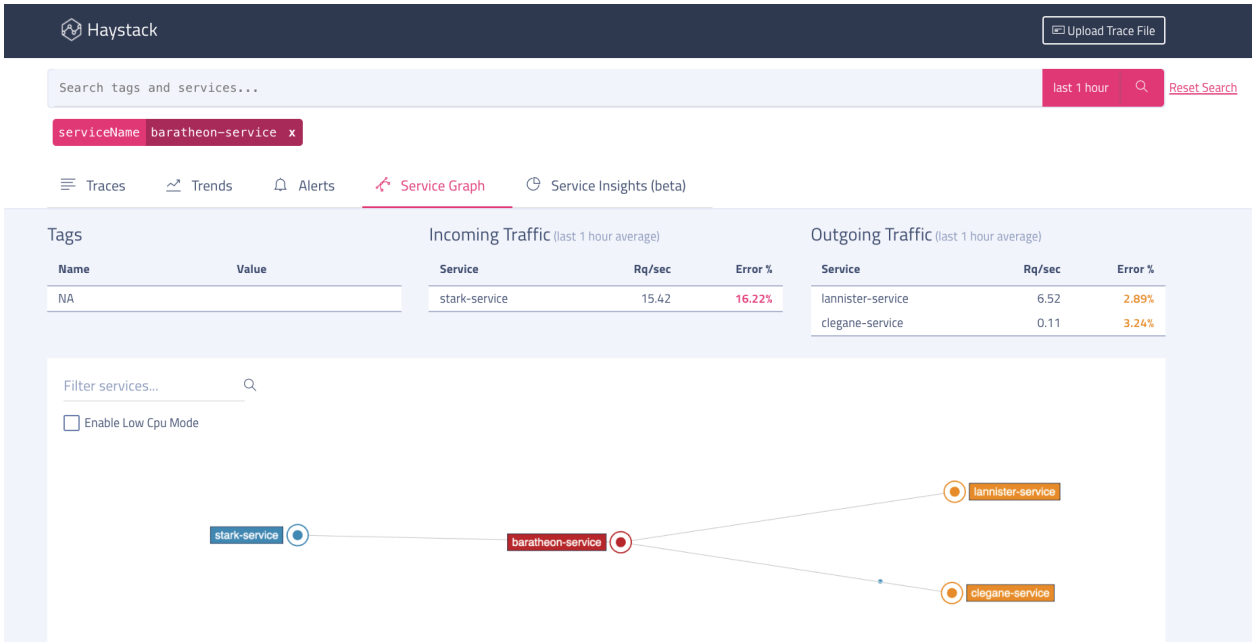

Fig. 2: Haystack Detailed View UI

2. [Microservice Graph Explorer] from Hootsuite
    a. The Microservice Graph Explorer is a webapp that allows users to view the relationship and status of microservices that implement the [Health Checks API]. The Health Checks API enforces the specification of the dependency relationship between services. In addition to displaying the health of individual services, the graph explorer also provides information regarding connection issues between services and the location of code, logs, and dashboards for services.
    b. No graph visualization.

Fig. 3: Microservice Graph Explorer UI

3. Manual Graph Generation Tools
    a. The following tools allow users to specify a dependency graph between services, but do not support live status checks.
    b. Service Dependency Graph
        i. Displays the request dependencies between services up to a given point in time. Users must manually register the dependencies by calling an API endpoint.

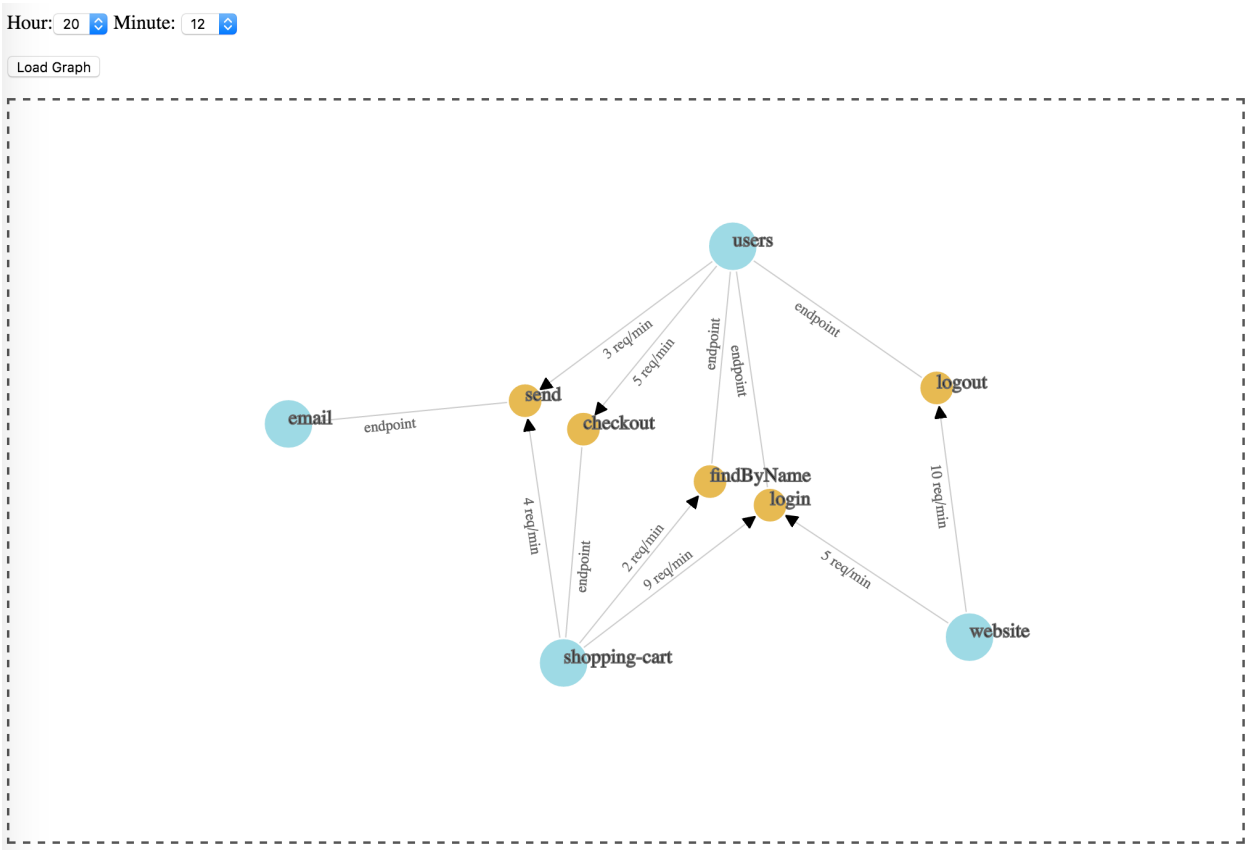

Fig. 4: Service Dependency Graph UI

    c. Octopus
        i. This project displays dependencies along with metadata for each service. Also relies on manual registration of dependencies via an API endpoint

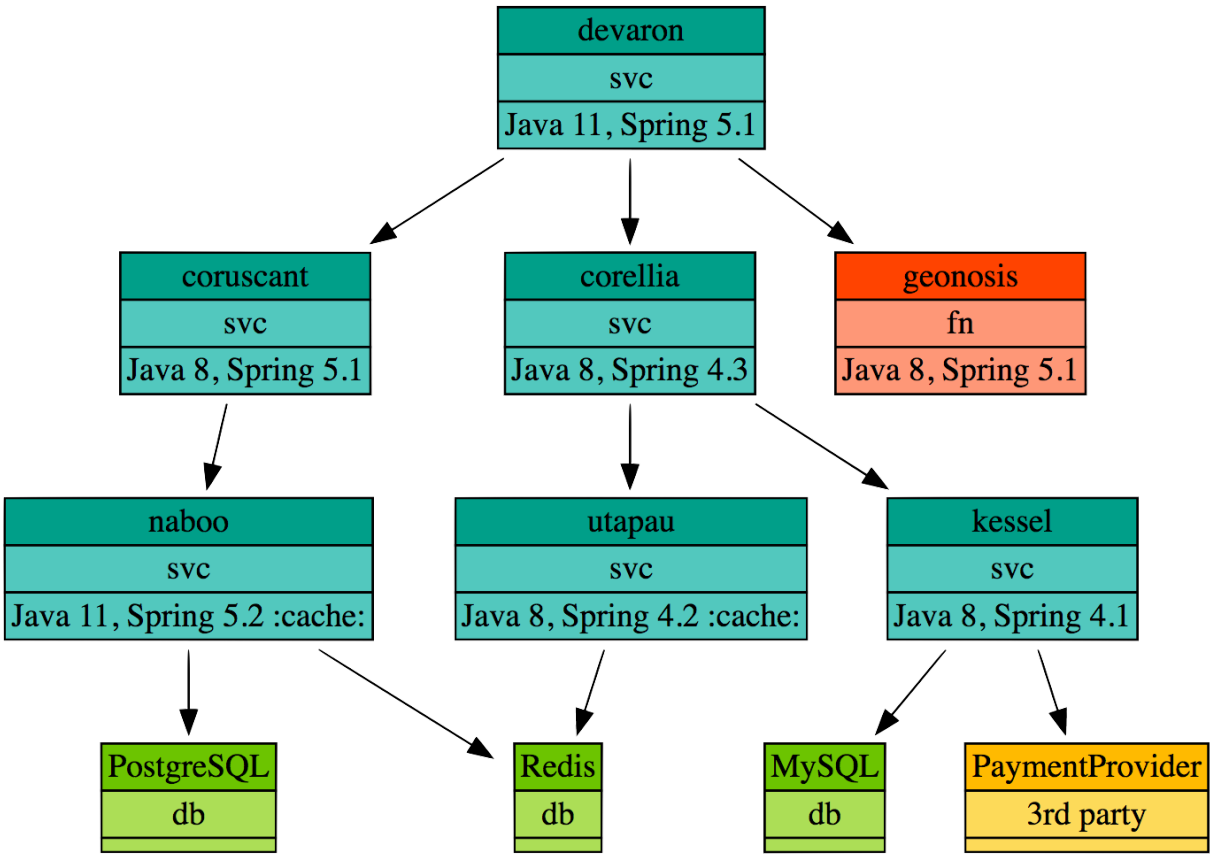

Fig. 5: Octopus UI

d. Vistecture

i. Displays dependencies configured in a YAML format.

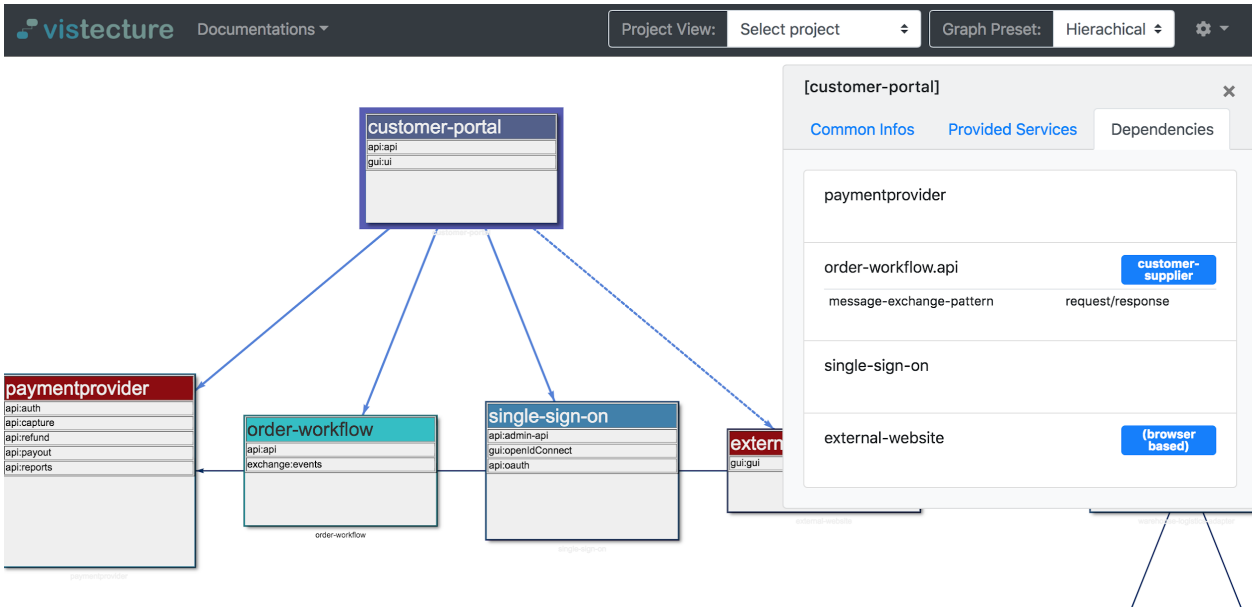

Fig. 6: Vistecture UI

However, none of the existing solutions provide the ability to investigate system status from a user journey-based granularity. This capability would be useful for PMs and engineers to identify issues impacting given user journeys, and, conversely, the user journeys that are affected by a given issue.

Requirements

The solution must provide a graphical representation of the dependencies between services. This visualization must allow users to:

- Display the overall status of a service
- Annotate the status and metadata (owner, comments, ...) of a service
- Display which services a given user journey relies on
- Display the status and SLIs of the service endpoints a given user journey relies on
- View the success rate of all user journeys
- Display which user journeys are affected by a specific service
- Easily integrate with external reporting server to provide dependency/SLI information
    - Expose API protos for reporting API to return
    - Reporting server could be an internal Google server or provided by users of the application

Design Ideas

A natural solution is to model the dependency relationships between services as a graph. Services and endpoints can be represented as nodes, while directed edges between nodes represent a dependency (e.g. pub/sub, producer/consumer, request/response, in Photos, typically an RPC). Each service can contain multiple endpoints. Each node can contain its own SLIs. Then, the dependencies for a given user journey can be identified by traversing the graph from the user journey's top level API calls (to be specified manually). Additional layers of abstraction (e.g. grouping services together into a "system" or "virtual node") are possible through the reuse of the node structure.

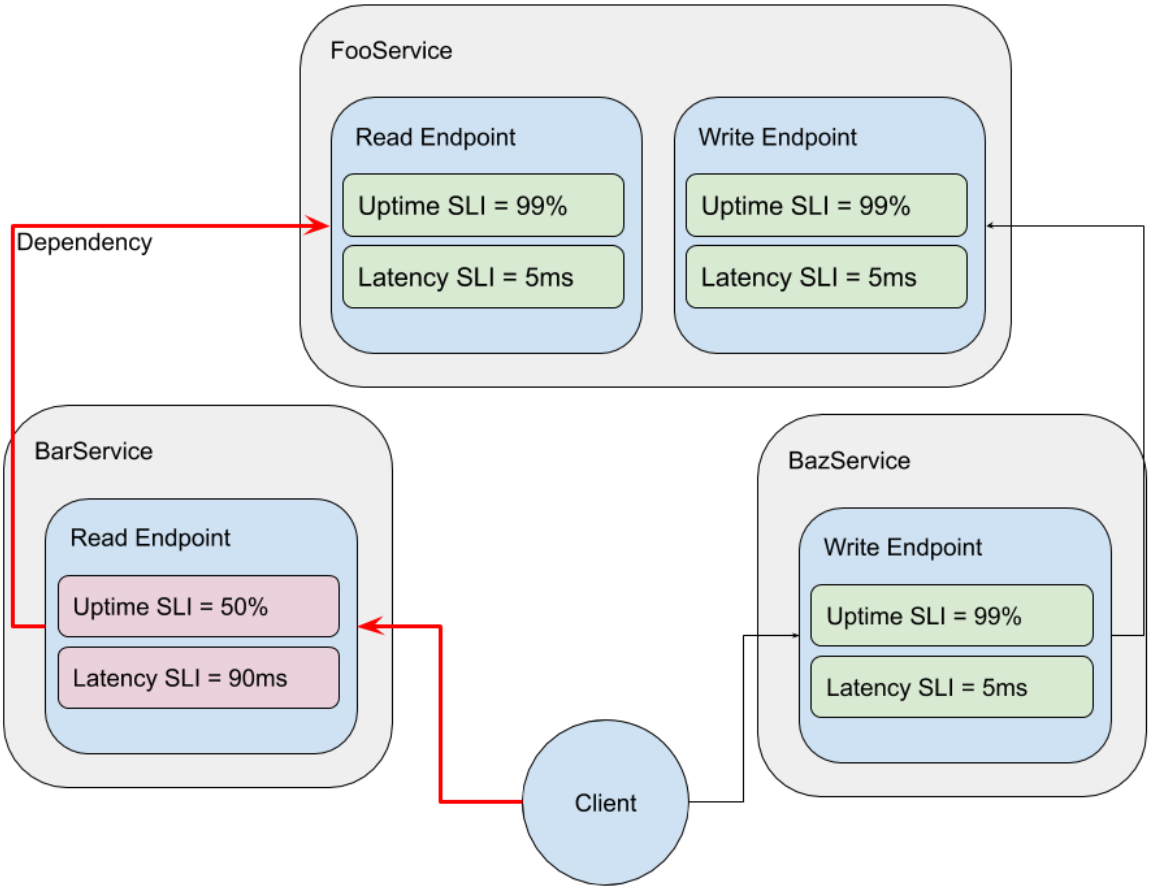

Fig. 7: Conceptual relationship between data structures

Data Structures

We choose to use protocol buffers to encode our data structures.

	Protocol Buffer Specification
Endpoint (a RPC procedure, a property within a service)	`message Node {` `  optional NodeType node_type = 1;` `  // name is a fully qualified, period delimited string (globally unique)` `  optional string name = 2;` `  repeated string child_names = 3;` `  optional string parent_name = 4;` `  // compute status from dependencies` `  optional Status status = 5;` `  repeated Dependency dependencies = 6;`

	`` // this endpoint's SLIs repeated SLI slis = 7; // can be extended to a full metadata message optional string comment = 8; } ```
Dependency (a graph edge)	``` message Dependency { optional string target_name = 1; optional string target_source_name = 2; optional string toplevel = 3; // if this dependency originates from a client optional string comment = 4; } ```
SLI (service level indicator)	``` message SLI { optional string service_name = 1; optional string endpoint_name = 2; optional SLIType sli_type = 3; optional double sli_value = 4; optional double slo_error_upper_bound = 5; optional double slo_error_lower_bound = 6; optional double slo_warn_upper_bound = 7; optional double slo_warn_lower_bound = 8; optional string comment = 8; } ```
Status	``` enum Status { STATUS_UNSPECIFIED = 0; STATUS_HEALTHY = 1; STATUS_WARN = 2; STATUS_ERROR = 3; } ```
SLIType	``` enum SLIType { SLITYPE_UNSPECIFIED = 0; SLITYPE_AVAILABILITY = 1; SLITYPE_LATENCY = 2; SLITYPE_THROUGHPUT = 3; } ```
UserJourney	``` message UserJourney { optional string name = 1; // a set of top level dependencies originating from Clients optional client_name = 2; optional Dependency dependencies = 3; } ```

Client	`message Client {` `  optional string name = 1;` `  repeated UserJourney user_journeys = 2;` `}`

For an initial prototype implementation, we may implement a simpler model by considering entire services as either up or down. To do this, we can simply specify dependencies at the Service granularity.

The tool will store each message in a dictionary for quick lookup and modification of its internal state.

Reporting API

The tool will need to talk to a reporting API to get information about the dependency topology, current system SLIs, and user journeys.

Reporting Service	`service ReportingService {` `  // for generating graph` `  rpc GetServices(ServiceRequest) returns (ServiceResponse) {}` `  // for updating statuses` `  rpc GetSLIs(SLIRequest) returns (SLIResponse) {}` `  // for user journeys` `  rpc GetClients(ClientRequest) returns (ClientResponse) {}` `  rpc SetComment(CommentRequest) returns (CommentResponse) {}` `}`
ServiceRequest	`message ServiceRequest {` `  // if name not provided, get all services` `  repeated string names = 1;` `  // not necessary for now` `  // optional ServiceMask = 2;` `}`
ServiceResponse	`message ServiceResponse {` `  repeated Service services = 1;` `}`
Additional request/responses not shown	`...`

For the initial version of this project, the tool will simply read local mock data instead of communicating with a server. Building upon this, we plan to implement a simple reporting server that serves static mock data, as a proof of concept and usage example. A tentative body of mock data can be found here.

In actual usage, the reporting server would need to be pre-configured with the dependency topology. Further investigation is required to determine how this server would collect the actual desired SLI metrics.

Dependency Graph Bootstrapping

The tool will request the set of Nodes and Clients at start time. By recursively tracing the edges defined by Dependencies, the tool will generate an internal graph representation of the service topology.
We will also supplement the set of Clients by loading locally saved protobufs. We can use the text_format module to persist human-readable protobuf values to disk. Thus, the initial set of Clients and UserJourneys can be manually created or modified by text editor for local testing, without the need to modify the central reporting API. In subsequent versions, the tool may provide an interface for adding, removing, and editing the initial Client and UserJourney protobufs.

SLI Updates

At a (user-specified?) interval, the tool will perform RPCs to fetch the SLIs of each endpoint from the central reporting API. The status/issue reporting API will return mock data for the purposes of this project. The reporting API responds with SLIs describing the current state of each endpoint within the service. We can then use this response to locally update the tool's internal state of each endpoint.

Status updates on (leaf) Nodes without dependencies can be done in any order after all SLIs have been updated, since their status depends only on their own SLIs.

However, in general, we need to propagate the status of the dependency chain downstream. We can do this performing depth first searches, starting at each top-level Client, to recursively traverse the graph. At each Node, we recursively check the status of the dependencies of the Node itself and its child Nodes, and update their respective statuses before popping a stack frame.

Status Computation

The status of a Node will be computed based on the status of its dependencies. If all dependencies are HEALTHY and all internal SLIs are within (slo_warn_lower_bound, slo_warn_upper_bound), then the status will be HEALTHY. If any dependency is WARN status

or the SLI is within (slo_error_lower_bound, slo_error_upper_bound), then the status will be WARN. The corresponding logic is used to compute ERROR status.

Language Choice

Python is simple and flexible, and has strong library support for networks, visualization, and UI construction. The author has more experience in Python, and it's generally easier to bootstrap applications and get started, given the (relatively) limited time frame.
Some library options include:

- NetworkX, a network analysis library
    - Limited visualization features
    - Provides graph analysis tools, algorithms
- Pyvis, a network visualization library (for local applications)
- Dash, a framework to build web applications for interactive data visualization with one of the following for graph visualization:
    - Plotly, a graphing library
    - Cytoscape, an interactive graph visualization library
- Standard testing libraries, pytest, unittest.mock, etc.

Given Python's simplicity and its rich library support, the author leans towards Python as the language of choice for this project.
We plan to use Dash to build the UI, and Cytoscape to provide the graph visualization features. Initially, the tool will handle processing of the internal representation of the graph. However, if it seems that too much code is dedicated to implementing graph algorithms, we may choose to integrate NetworkX to handle said responsibility.
Cytoscape has support for compound nodes, which model the dependency graph relationship between Services and Endpoints naturally. Creating a web application with Dash seems beneficial for the tool to be more widely accessible and available. Moreover, the ease of constructing network interactivity features with Cytoscape is also attractive.

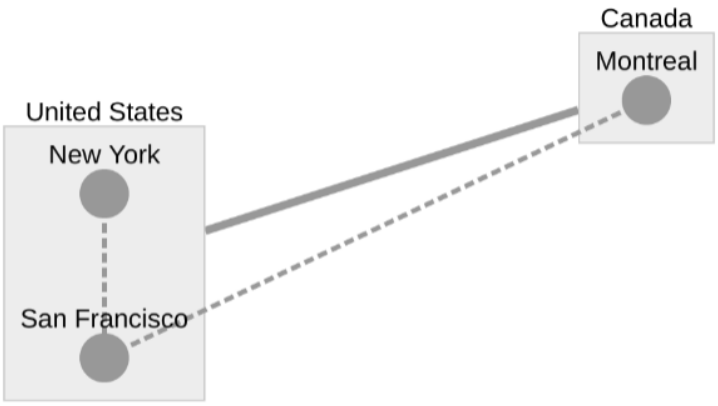

Fig. 8: Cytoscape Compound Nodes

Web UI

We plan to use Dash, which spins up a Flask server and allows us to use React components in Python to build out a Web UI. For individual use, users could run the app locally but this could be containerized and deployed to GCP or some similar service. We do not foresee the need to develop custom UI components; the bulk of the work is connecting existing components and adding interactivity to display our data.

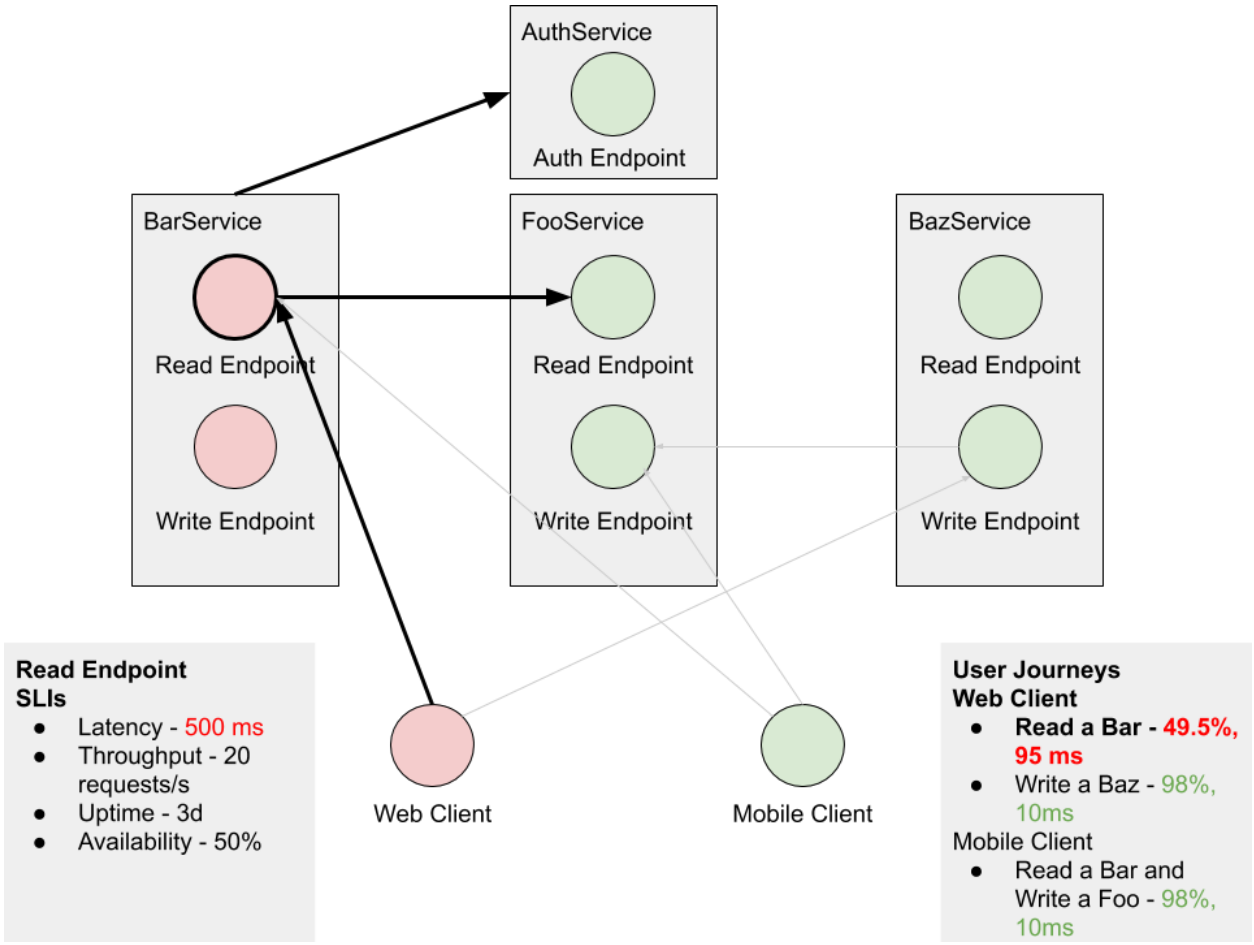

Fig. 9: UI with a User Journey and Endpoint Selected

The UI consists of three main elements, the graph (middle), the endpoint panel (left), and the user journey panel (right). When a user clicks a User Journey from the right panel, the dependency path through the graph is highlighted while unused dependencies are greyed out. When an endpoint node in the graph is clicked, the endpoint panel on the left is updated with the SLI of the endpoint. In the figure above, the “Read a Bar” User Journey is selected, along with the BarService.Read endpoint.

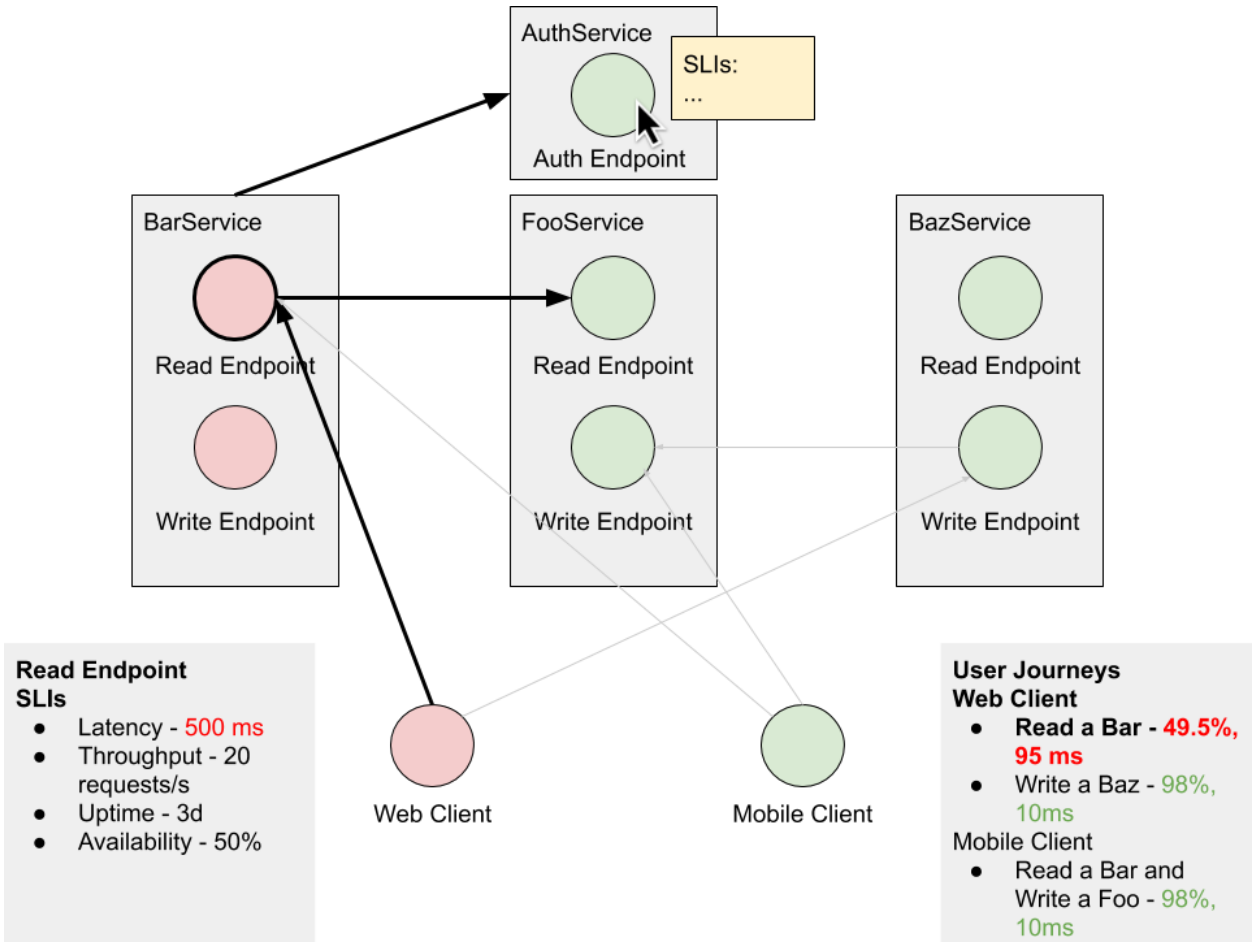

Fig. 10: UI Hover

When the user hovers over a Node, a temporary SLI panel will appear and display the Node's SLI values.

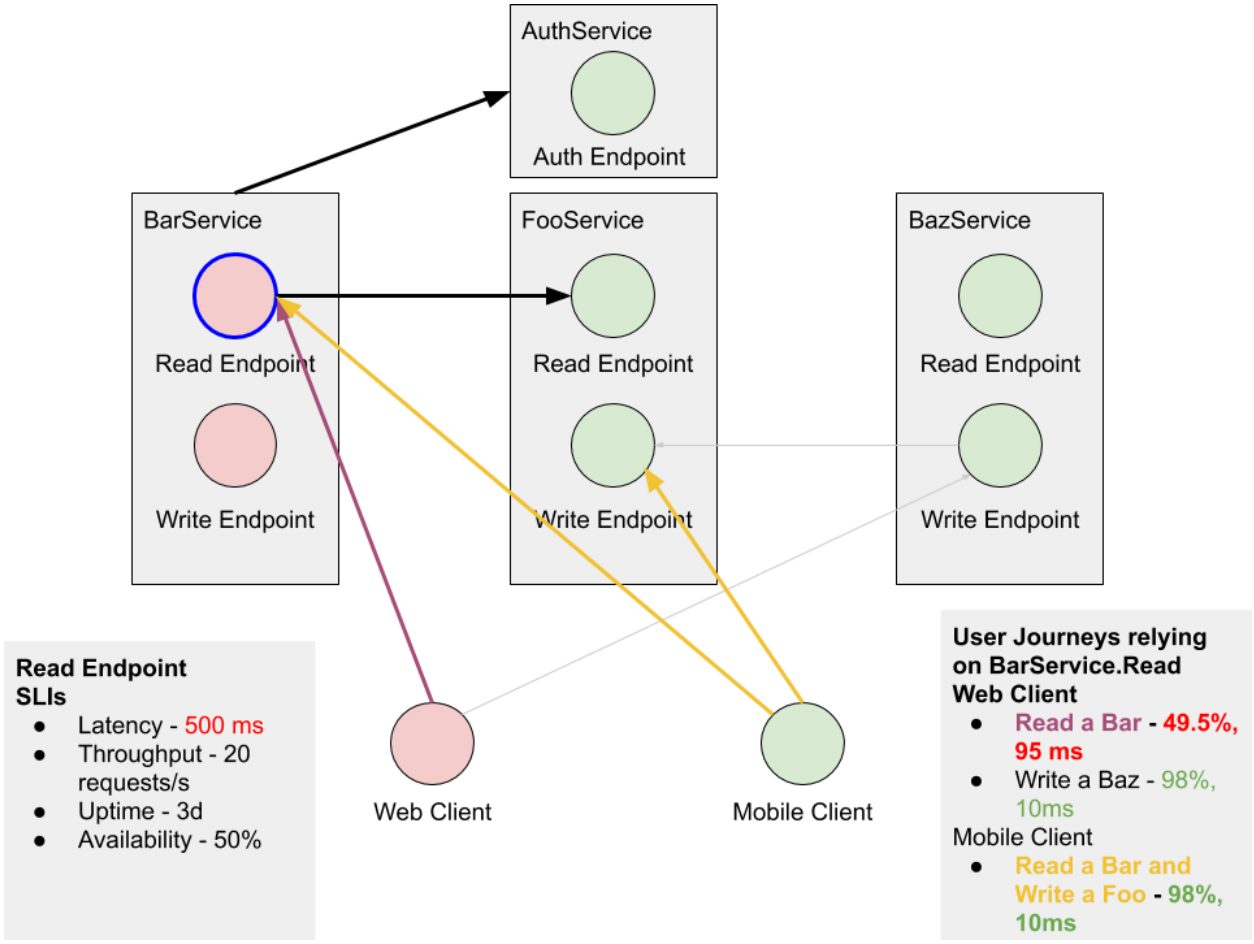

Fig. 11: Viewing User Journeys relying on a single Node

When an endpoint is double-clicked, all user journeys that depend on the endpoint are highlighted and the dependencies are bolded. The mockup in the figure above illustrates the tool displaying user journeys that rely on the BarService.Read endpoint. The dependencies for each user journey are color coded, while shared dependencies (downstream from the selected endpoint) remain black.

Nodes are also draggable and can be rearranged to the user's preference.

Alternatives Considered

Language Choice

Compared to Python, Java offers additional type safety, may be more performant, and is also more accessible to members of the Photos Infra team. However, it's often more verbose and less familiar to the author.

- JGraphT, a graph library
    - Requires JGraphX for visualization
- GraphStream, a graph library with interactivity

- JUNG, a graph library (seems outdated)

The author leans away from Java due to his preference for Python and its comprehensive support for UI construction.

Dependency Graph Bootstrapping

Alternatively, we can perform static code/build file analysis to determine the dependency relationship between endpoints. However, this would likely limit the tool's support for services that may be built or implemented differently (compared to the Photos codebase).

We could also consider allowing users to specify the dependency relationship through local configuration files. These files would have to be manually updated, and may quickly become out of sync with the underlying services that they describe, by virtue of being located externally from the service's codebase.

However, it ultimately seems most natural for the central reporting API to specify the dependencies that its endpoints rely on. That way, maintainers can more easily keep the dependencies up to date.

Privacy Considerations

The user journey troubleshooting tool should not access any user information, whether in aggregate, anonymized, or otherwise. The implementation described above operates independently from each service's functionality and data, and relies only on the each service's own status API to expose information about endpoints and SLIs. The only persisted data are the User Journey protobufs. These protobufs should not expose any information beyond the endpoints of the system being monitored.

Stretch Goals

Scalability and Deployment

With Dash, all state is stored in the web browser and the backend is stateless. Thus, multiple users can simultaneously access the tool. Dash apps can be containerized and deployed with Kubernetes on GCP, with a load balancer to scale the application.

As an open source project on GitHub, we can set up GitHub Actions for Python for convenient CI/CD for the tool.

As the number of Clients and User Journeys grows, it may be unsustainable to manually edit and store protobuf objects in local files. We may choose to save our protobufs in a database (ProfaneDB?) to support the storage and retrieval patterns at scale.

Internationalization and Accessibility

By including name and comment string fields in Service, Endpoint, Dependency, and SLI protobufs, the tool enables users to specify human readable descriptions of the parts of the system.
However, the protobuf enums use English-specific terms for their values (e.g. “HEALTHY”, “LATENCY”). Moreover, the protobuf field names are English-specific as well, which may pose difficulties for users attempting to manually define the base set of Clients and UserJourneys.

A possible accessibility goal can be to include a colorblind mode setting, to replace the conventional red/yellow/green traffic light for status indication. Other features could include keyboard-only usage mode or compatibility with screen reading software.
Dash is automatically tested against Chrome and is manually tested on an array of other browsers. By virtue of being a web app, the tool would be accessible to a wider base of users, as it would not be required to manually clone and run the source locally. Dash and Cytoscape are also compatible with mobile devices.