Synapse - A Remediation System for Enterprise Ticketing

Problem

In Enterprise IT - a large class of problem gets repeated daily - and their remediation sometimes changes frequently or infrequently. The larger the size of the enterprize is, the larger the day to day problems in IT. While Enterprise ticketing systems are robust - and there are plenty of players in the domain (Salesforce, ServiceNow, SAP, Remedy, ZenDesk, FreshDesk to name a few) - the actual remediation is still done mostly by human agents. Some companies are bringing the power of analytics and AI to solve some of the problems (notably : Simplifai, Freshdesk etc) but overall the field is dominated by human agents.

Solution Idea

User Story

Before

1. User face some problem.
2. Opens ticket / send mail / slack / call
3. Support agent picks up the ticket (create a ticket)
4. Assign it to the apropriate team
5. Someone from the team picks up the ticket
6. Connects with the user and try solving the issue
7. If solved, closes the ticket

After

1. User faces a problem
2. User chats it to Synapse - via email or chat
3. Synapse creates a ticket - assignes the team - also updates possible remediation
4. User accepts or rejects the resolution
5. Resolution team picks up the ticket with updated information and solves the problem for the user
6. If solved, closes the ticket

Algorithms

Reccomendation

Given past data and a new ticket predict the following:

1. What sort of past tickets having similar problems
2. What sort of past tickets can be used to solve the the problem

Similarity of Tickets

Problem [1] gives in to the ticket similarity problem. It can be formally defined as :

Ticket has ID, Title, and text Description. Find out which past tickets are similar to the current one.

Similarity of Solutions

Problem [1] gives in to the ticket resolution similarity problem. It can be formally defined as :

Ticket has ID, Title, and text Description and resolution steps.

1. Find out which past tickets are similar to the current one.
2. Which group/groups the ticket would be classified into
3. Which remediation steps will be useful to solve the current one

Categorization of Problems

The following questions needs to be answered:

1. What sort of "rough" groups of "root causes" are there in the system?
2. How can we classify different issues
3. What are the distribution of root causes over issues?
4. How the root causes are changing over time ?
5. What sort of issues will be a problem in the near future?

Data Scraping

Synapse ran over past data - and thus primary problem for the system is to gather as much data as possible. Before any integration - the system needs to learn from huge amount of past data - from at least 2~3 years of data - that is at least 10 million reccords. Post integration the system can smartly pick up the data added up - and train itself.

Initial load of 10 million reccords is known as boot strapping. Post integration it can learn near real time.

Data Schema

```
{  /* every field is string */
    "id" : "ticket id", // mandatory
    "title" : "title of the issue", // mandatory
    "description" : "description of the issue",  // mandatory
    "opened_at" : "date time when the ticket was opened" , // optional
    "closed_at" : "date time when the ticket was opened" , // optional
    "opened_by" : "user who opened" , // optional
    "closed_by" : "user who closed",  // optional
    "category" : "category of the issue" // optional
    /* one can add more fields - it won't matter */
}
```

Boot Strapping

Initial boot strapping for the data was a manual effort. One has to export the data in the TSV or Excel format - and from there it would be tranformed into the data schema format, in SPARK Json files - where each line of the file is a JSON reccord. This tended to take at least 1 day - or more based on the system from where the data is being exported. Languages for data transformation varied - ranging from ZoomBA, PowerShell, and in some cases Python if the system supported.

Near Real Time

Once the system integration was complete - any new ticket via event hook will be automatically dumped upon closure to a specific folder - from where a batch job will pick it up and then merge them into a training file having similar structure as boot strapping. The system merely now had to learn from the new data dump.

Integration Points

System connects with various other systems in the Enterprise Ecosystem:

1. Auth System (primarily LDAP)
2. Communication
    1. Mail
    2. Slack
    3. SMS
3. Ticketing System - for training and data
4. Knoweldge Articles and SOPs - for training and data

Components

Core Engine

1. Algorithm Engine - various algorithm for recomendation and classification
2. Data Processeor - loads data from SPARK JSON files and connects with data store
3. Data Store - Cassandra for storing the trained model data
4. Web API

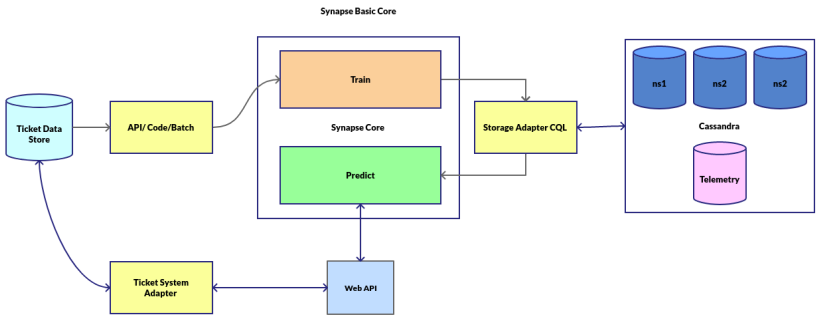
```
Method : "POST"
URL : "/synapse/query"
REQ BODY: {
  "ns" : "namespace - context for recomendation"
  "i" : "id of the ticket" , // optional
  "t" : "title of the ticket", // optional
  "d" : "description of the ticket" // optional
}
/* t or d is enough */

RESP BODY: {
    [
        { /* returning matching old tickets with score */
```

```
            "s" : "score of the ticket",
            "i" : "Id of the ticket",
            "t" : "title of the ticket",
            "d" : "Description of the ticket",
            ... // more optional field if trained
        }
       ,... /* same schema */
    ]
}
```

One Synapse core engine was capable of maintaining multiple namepspaces to give sharding and data isolation. Namespaces makes the following trivial:

1. Data Isolation : Different categories of data can be isolated
2. Parallelism : Can shard multitude of data on different namespace and then finally parallel accumulation is possible
3. Can specifically "search" different time frames stored in differeent namespaces



JOLT

Enterprise integration is a lot of hard work and a lot of integration code. It is both push and pull. To aid in these - a tiny infrastructure server was created with following deign goals:

1. Configuration Driven
2. In built proxy capability
3. Fast
4. Pluggable Auth
5. Drop Files and create End Points
6. Timers & Cron capabilities

Spark-Java was used as the core engien for this. It had JSR-223 support, via which all possible JVM languages could be used to write logic.

Custom DSL like ZoomBA were used to intepret configurations, data massaging as well as coding for the end points. A Typical JOLT configuration would like this:

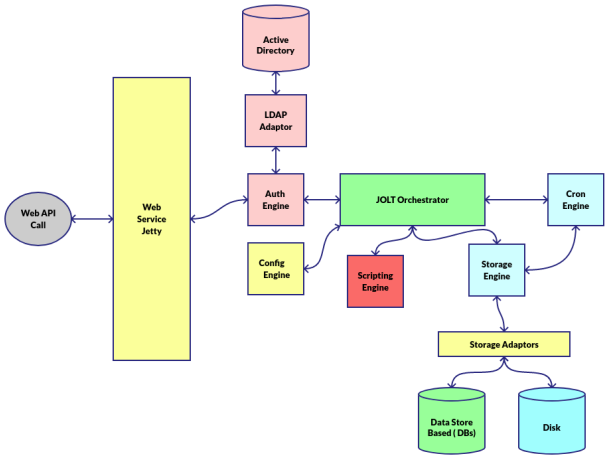
```
home : /foo/bar
port: 8080
routes:
   get:
     /hello :  _/hello.zm
   post:
     /q : _/query.zm

auth: _/auth.yaml
```

Default request response was JSON with gzip compression turned on.

A typical JOLT route mapper (read script that will be called when a route gets matched with the request) is as follows:

```
/*
req : request parameter
resp: will be automatically responded after transformin into JSON
*/
val params = req.params
/* do something with it.. */
val res = Map("r1" -> "Result1", "r2" -> "Result2")
return res // respond back
```



All possible applications were written on top of JOLT - and hence were JOLT apps. The team could enable 150 functionalities in 3 months due to it's extensibility and capability.

Automation Server

Many cases the remediation would require human agents to run a script on the effected users machine. To aid to, an automation server was created. This server hosts meta information in PGSQL as follows:

```
{
    "id" : "id of the script",
```

```
    "auth" : "auth group to run it",
    "desc" : "description",
    "version" : "version of the script",
    "path" : "shared path where the script is located",
    "checksum" : "for verification that the script content is legit",
    "params" : [ // arguments for the script
        {
            "name" : "name of the arg",
            "desc" : "description",
            "def" : "default value"
        }
    ]
}
```

This automation server is a JOLT app. It exposes API's for CRUD for the Automation Scripts along with Run. While primary host for the automation server is assumed to be Windows (Enterprise IT is windows) - Mac and Linux were also supported.

The exposed API's were as follows:

```
Method : GET
Path: /all/[ug]
Response:
[
    // all scripts available in the server
]
Optional : UG UserGroup
Only user group scripts will be shown
```

```
Method: POST
Path: /start/<script_id>
Body: {
    "params" : [ args... ],
    "user_token" : "user token"
}
Response:
{
    "run_id" : "globally unique run id for the script",
    "status" : "ok|fail"
}
```

```
Method: GET:
Path : /inspect/<run_id>
Response:
{
    "script_id" : "which script",
    "start_time" : "when started",
    "end_time" : "if available",
    "success" : "true|false",
    "args" : [],
    "out" : "", // standard output dump
    "err" : "", // standard error dump
    "user" : "who triggerd it"
}
```

```
Method: GET
Path: /history/<script_id>/[from]/[to]
Response :
{
    [ {
        "run_id" : "id for the run",
        "args" : [ ],
        "time" : "when ran",
        "user" : "who ran",
        "err" : "error",
        "out" : "output",
        "code" : "exit code",
    },... //many more
     ]
}
```

Adapters

Any system integration will be writing adapters. A JOLT app solves that problem via injecting "data massaging" scripts from any JSR-223 languages. These were used to gather data from multiple data sources of varied schema to match into the Synapse, while pushing data from Synapse system to the various other enterprise software entities.

Batch

System supported running BATCH jobs - due to existence of timers and cron capabilties in JOLT. All Batch jobs were stored just like automation scripts, but a meta batch table was used to trigger jobs automatically from them.

```
{
    "job_id" : "id of the job",
    "owner" : "owner id for the job",
    "published" : "date of publication",
    "modified" : "last modified date",
    "script_id" : "which script to run",
    "args" : [ ] , // args to pass to the scripts
    "cron" : "cron type timing for the job",
    "decription" : "what is this job?"
}
```

Auth

System supported pluggable auth models. Most used models were using LDAP to connect the users to the organizations active directory. The auth system would verify once the user identity against the Active Directory - and issue an expirable token and kept on using that token. The storage used were PGSQL for the storgae of the token. For issueing the token JOLT App was used. Proposal was eventually to move into JWT.

Auth integration was done via Adapters. LDAP user data was synchronized via batch capabilties.

Admin Users could create roles and assignes users into roles.

Event Hooks

Many Systems like ServiceNow and Slack let's us use API Callback (webhooks) to integerate systems. That essentially means, again, booting up a tiny webserver and registering for the approrpriate payloads and consuming them.

In this methodology the following were done:

1. Ticketing Systems WebHooks
    1. Getting new ticket data
    2. getting ticket data when tickets are being closed
    3. Other Update on tickets
    4. Rejection of Synapse response on ticket
2. Communication Systems Hooks
    1. New Communciation - Slack / Mail
    2. Update into a channel - new events

Notifier

Notification system was a way to gather all possible events aggregated from the event hook to generate a summary communication to the system. We supported sending mail, slack messages, posting into the ticketing system as well as sending SMS messages and voice calls via Twilio.

Twilio system was proxied via another JOLT app.

Metrics

Each individual components kept on log files - for example JOLT will have their own log files. Other systems will put various key state changes in their own storage - which then was seen using Graphana.

There were adapters which modified the log files into CSV format to load back into Graphana.

Notifiers were used in mail / corporate slack / sms to send key events on Synapse systems if the system fails for some reason.

Users

There were 4 categories of users in Synapse.

Vanilla

These are the users who are end users of the system who would simply get automatic reccomendation for any issue they are trying to create.

Support Agents

These users will use Synapse UI to figure out custom solutions for the tickets for users.

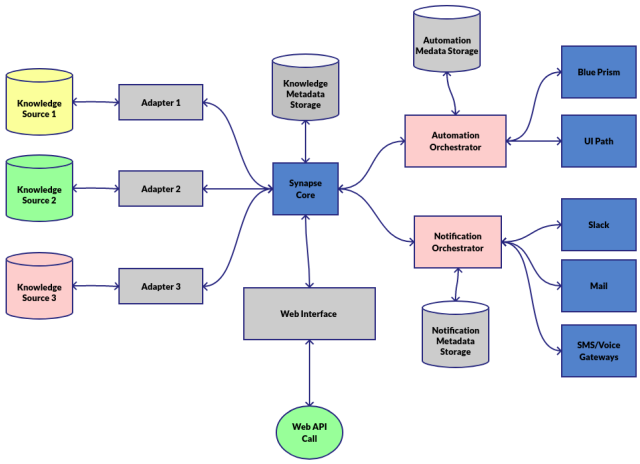
Support Leads

These had access to Synapse metrics and dashboard (readonly) to see if the system is working as expected. They also would check on performance metrics of Synapse, merely how many tickets were resolved using the Synapse reccomendation. They can add agents to groups and such.

System Admins

They had access to the underlying infra for the whole system - such as every configuration and database and tables. They could configure users, and roles and groups, and assign users into them.

Uber System Diagram



References

1. JSR 223 - https://en.wikipedia.org/wiki/Scripting_for_the_Java_Platform
2. Spark Java - https://sparkjava.com/documentation

Patents

1. https://patents.justia.com/patent/20210342536
2. https://patents.justia.com/patent/20210365810

Papers

1. https://arxiv.org/abs/1604.05903
2. https://arxiv.org/abs/2011.13832