# app.py (Enterprise-Grade Query Parser with Gemini LLM)

import streamlit as st

import pandas as pd

import json

import re

import google.generativeai as genai

from typing import Dict, Any, List, Tuple, Optional

from enum import Enum

from difflib import SequenceMatcher

import datetime

from dateutil import relativedelta

import logging

from dataclasses import dataclass

import asyncio

import time

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

class AgentType(Enum):

QA = "qa\_agent"

INSIGHT = "insight\_agent"

WHAT\_IF = "what\_if\_agent"

RECOMMENDATION = "recommendation\_agent"

class EntityType(Enum):

PROJECT = "project"

EDL = "edl"

LOCATION = "location"

TIME\_PERIOD = "time\_period"

METRIC = "metric"

ACTION = "action"

@dataclass

class Entity:

type: str

raw\_value: str

resolved\_value: Optional[str] = None

confidence: float = 0.0

resolution\_method: str = "unknown"

alternatives: List[str] = None

position: int = 0

class GeminiService:

"""Enterprise-grade Gemini service for Colab environment"""

def \_\_init\_\_(self, api\_key: str):

self.api\_key = api\_key

genai.configure(api\_key=api\_key)

self.model = genai.GenerativeModel('gemini-pro')

def extract\_entities\_with\_retry(self, query: str, entity\_context: Dict, max\_retries: int = 3) -> List[Dict]:

"""Use Gemini to extract entities with retry logic"""

prompt = f"""

You are an expert entity extraction system for workforce analytics. Extract ALL entities from the user query.

AVAILABLE ENTITY TYPES:

- PROJECT: Project names, accounts, clients (e.g., "Google Street View", "Waymo Rider")

- EDL: Manager names, EDLs, leadership (e.g., "Ravikumar", "John Smith")

- LOCATION: Geographic locations (e.g., "India", "Philippines", "Offshore")

- TIME\_PERIOD: Time references (e.g., "last month", "Q2", "YTD")

- METRIC: Business metrics (e.g., "utilization", "NBL", "attrition")

- ACTION: Actions and verbs (e.g., "reduce", "compare", "show")

KNOWN ENTITIES IN OUR SYSTEM:

Projects: {list(entity\_context.get('projects', []))[:10]} # Show first 10

EDLs: {list(entity\_context.get('edls', []))[:10]}

Locations: {list(entity\_context.get('locations', []))}

INSTRUCTIONS:

1. Extract ALL entity mentions from the query

2. Classify each entity into one of the above types

3. Return the exact text span from the query

4. For ambiguous references, provide your best guess

5. Return ONLY valid JSON, no other text

QUERY: "{query}"

Return JSON format:

{{

"entities": [

{{

"type": "project|edl|location|time\_period|metric|action",

"value": "exact text from query",

"position": starting\_character\_position

}}

]

}}

"""

for attempt in range(max\_retries):

try:

response = self.model.generate\_content(prompt)

# Extract JSON from response

response\_text = response.text.strip()

if response\_text.startswith('```json'):

response\_text = response\_text[7:-3] # Remove ```json and ```

result = json.loads(response\_text)

return result.get("entities", [])

except Exception as e:

logger.warning(f"Gemini entity extraction attempt {attempt + 1} failed: {e}")

if attempt < max\_retries - 1:

time.sleep(2 \*\* attempt) # Exponential backoff

else:

logger.error("All Gemini entity extraction attempts failed")

return []

def route\_to\_agent\_with\_retry(self, query: str, entities: List[Dict], max\_retries: int = 3) -> Dict[str, Any]:

"""Use Gemini to determine the best agent with retry logic"""

entity\_summary = "\n".join([f"- {e['type']}: {e['value']}" for e in entities])

prompt = f"""

STRICTLY classify this workforce analytics query into ONE of these FOUR agent types:

QA\_AGENT: Factual questions, metrics, comparisons, lists

Examples: "current utilization", "top projects", "compare EDLs"

INSIGHT\_AGENT: Analysis of trends, root causes, explanations

Examples: "why did utilization drop", "explain high NBL", "what's driving attrition"

WHAT\_IF\_AGENT: Scenario simulations, impact analysis, hypotheticals

Examples: "what if I reduce 5 NBL", "impact of moving resources", "simulate attrition"

RECOMMENDATION\_AGENT: Action plans, strategies, optimization

Examples: "how to improve utilization", "recommend NBL strategy", "achieve 85% target"

EXTRACTED ENTITIES:

{entity\_summary}

QUERY: "{query}"

Return ONLY this JSON format:

{{

"agent\_type": "qa\_agent|insight\_agent|what\_if\_agent|recommendation\_agent",

"confidence": 0.0-1.0,

"reasoning": "brief explanation"

}}

"""

for attempt in range(max\_retries):

try:

response = self.model.generate\_content(prompt)

# Extract JSON from response

response\_text = response.text.strip()

if response\_text.startswith('```json'):

response\_text = response\_text[7:-3]

result = json.loads(response\_text)

return result

except Exception as e:

logger.warning(f"Gemini agent routing attempt {attempt + 1} failed: {e}")

if attempt < max\_retries - 1:

time.sleep(2 \*\* attempt)

else:

logger.error("All Gemini agent routing attempts failed")

return {"agent\_type": "qa\_agent", "confidence": 0.5, "reasoning": "Fallback due to LLM failure"}

class EntityResolver:

"""Handles fuzzy matching and entity resolution"""

def \_\_init\_\_(self, prism\_data: pd.DataFrame):

self.prism\_data = prism\_data

self.entity\_cache = {}

self.\_build\_entity\_index()

def \_build\_entity\_index(self):

"""Build indexes of all known entities for fuzzy matching"""

self.project\_names = set(self.prism\_data['project\_name'].dropna().unique())

self.rev\_mappings = set(self.prism\_data['rev\_mapping'].dropna().unique())

self.edl\_names = set(self.prism\_data['edl\_name'].dropna().unique())

self.locations = set(self.prism\_data['utilization\_location'].dropna().unique())

# Common nicknames and abbreviations

self.nickname\_map = self.\_build\_nickname\_map()

def \_build\_nickname\_map(self) -> Dict[str, str]:

"""Build mapping of common nicknames to full names"""

nickname\_map = {}

for edl in self.edl\_names:

if pd.notna(edl):

# Extract first name for common nicknames

first\_name = edl.split()[0] if ' ' in edl else edl

nickname\_map[first\_name.lower()] = edl

# Common nickname patterns

if first\_name.startswith('Ravi'):

nickname\_map['ravi'] = edl

elif first\_name.startswith('John'):

nickname\_map['johnny'] = edl

# Add more patterns as needed

return nickname\_map

def fuzzy\_match(self, input\_text: str, candidates: set, threshold: float = 0.6) -> List[Tuple[str, float]]:

"""Fuzzy match input against candidate strings"""

matches = []

input\_lower = input\_text.lower()

for candidate in candidates:

if pd.isna(candidate):

continue

candidate\_lower = candidate.lower()

# Exact match

if input\_lower == candidate\_lower:

matches.append((candidate, 1.0))

continue

# Partial match

if input\_lower in candidate\_lower or candidate\_lower in input\_lower:

matches.append((candidate, 0.9))

continue

# Fuzzy string matching

similarity = SequenceMatcher(None, input\_lower, candidate\_lower).ratio()

if similarity >= threshold:

matches.append((candidate, similarity))

return sorted(matches, key=lambda x: x[1], reverse=True)

def resolve\_project(self, project\_ref: str) -> Dict[str, Any]:

"""Resolve project reference with fuzzy matching"""

# Check nickname map first

if project\_ref.lower() in self.nickname\_map:

resolved = self.nickname\_map[project\_ref.lower()]

return {"resolved\_name": resolved, "confidence": 0.95, "method": "nickname"}

# Try exact matches first

if project\_ref in self.project\_names:

return {"resolved\_name": project\_ref, "confidence": 1.0, "method": "exact"}

if project\_ref in self.rev\_mappings:

return {"resolved\_name": project\_ref, "confidence": 0.9, "method": "rev\_mapping"}

# Fuzzy matching

project\_matches = self.fuzzy\_match(project\_ref, self.project\_names)

rev\_matches = self.fuzzy\_match(project\_ref, self.rev\_mappings)

all\_matches = project\_matches + rev\_matches

if all\_matches:

best\_match = all\_matches[0]

return {

"resolved\_name": best\_match[0],

"confidence": best\_match[1],

"method": "fuzzy",

"alternatives": [match[0] for match in all\_matches[1:3]]

}

return {"resolved\_name": None, "confidence": 0.0, "method": "none"}

def resolve\_edl(self, edl\_ref: str) -> Dict[str, Any]:

"""Resolve EDL reference with fuzzy matching"""

# Check nickname map

if edl\_ref.lower() in self.nickname\_map:

resolved = self.nickname\_map[edl\_ref.lower()]

return {"resolved\_name": resolved, "confidence": 0.95, "method": "nickname"}

# Exact match

if edl\_ref in self.edl\_names:

return {"resolved\_name": edl\_ref, "confidence": 1.0, "method": "exact"}

# Fuzzy matching

matches = self.fuzzy\_match(edl\_ref, self.edl\_names)

if matches:

best\_match = matches[0]

return {

"resolved\_name": best\_match[0],

"confidence": best\_match[1],

"method": "fuzzy",

"alternatives": [match[0] for match in matches[1:3]]

}

return {"resolved\_name": None, "confidence": 0.0, "method": "none"}

def resolve\_location(self, location\_ref: str) -> Dict[str, Any]:

"""Resolve location reference"""

location\_ref = location\_ref.lower()

# Common abbreviations

location\_abbr = {

'ph': 'philippines', 'ind': 'india', 'us': 'usa', 'usa': 'usa',

'offshore': 'offshore', 'onsite': 'onsite'

}

if location\_ref in location\_abbr:

resolved = location\_abbr[location\_ref]

if resolved in [loc.lower() for loc in self.locations]:

return {"resolved\_name": resolved.title(), "confidence": 0.9, "method": "abbreviation"}

# Exact match

if location\_ref in [loc.lower() for loc in self.locations]:

return {"resolved\_name": location\_ref.title(), "confidence": 1.0, "method": "exact"}

return {"resolved\_name": None, "confidence": 0.0, "method": "none"}

class TimePeriodExtractor:

"""Extract and normalize time periods"""

def \_\_init\_\_(self):

self.current\_date = datetime.datetime.now()

def extract\_time\_period(self, time\_ref: str) -> Dict[str, Any]:

"""Extract and normalize time period references"""

time\_ref = time\_ref.lower()

# Month references

month\_patterns = {

r'last month': self.\_get\_last\_month,

r'this month': self.\_get\_this\_month,

r'next month': self.\_get\_next\_month,

r'current month': self.\_get\_this\_month,

}

# Quarter references

quarter\_patterns = {

r'q[1-4]': self.\_resolve\_quarter,

r'quarter [1-4]': self.\_resolve\_quarter,

}

# Relative time periods

relative\_patterns = {

r'past (\d+) months?': self.\_get\_past\_months,

r'last (\d+) months?': self.\_get\_past\_months,

r'ytd': self.\_get\_ytd,

r'year to date': self.\_get\_ytd,

}

# Check patterns in order of specificity

for pattern, resolver in {\*\*relative\_patterns, \*\*month\_patterns, \*\*quarter\_patterns}.items():

match = re.search(pattern, time\_ref)

if match:

result = resolver(match)

if result:

return result

return {"period": None, "confidence": 0.0, "method": "none"}

def \_get\_last\_month(self, match) -> Dict[str, Any]:

last\_month = self.current\_date - relativedelta.relativedelta(months=1)

return {

"period": last\_month.strftime("%Y-%m"),

"display": last\_month.strftime("%B %Y"),

"confidence": 0.95,

"method": "last\_month"

}

def \_get\_this\_month(self, match) -> Dict[str, Any]:

return {

"period": self.current\_date.strftime("%Y-%m"),

"display": self.current\_date.strftime("%B %Y"),

"confidence": 0.95,

"method": "this\_month"

}

def \_get\_past\_months(self, match) -> Dict[str, Any]:

months = int(match.group(1))

periods = []

for i in range(months):

period = self.current\_date - relativedelta.relativedelta(months=i+1)

periods.append(period.strftime("%Y-%m"))

return {

"period": periods,

"display": f"Past {months} months",

"confidence": 0.9,

"method": "past\_months"

}

def \_get\_ytd(self, match) -> Dict[str, Any]:

start\_month = f"{self.current\_date.year}-01"

current\_month = self.current\_date.strftime("%Y-%m")

return {

"period": [start\_month, current\_month],

"display": f"YTD {self.current\_date.year}",

"confidence": 0.95,

"method": "ytd"

}

class EnterpriseQueryParser:

"""Enterprise-grade query parser with Gemini LLM integration"""

def \_\_init\_\_(self, prism\_data: pd.DataFrame, gemini\_api\_key: str):

self.entity\_resolver = EntityResolver(prism\_data)

self.time\_extractor = TimePeriodExtractor()

self.gemini\_service = GeminiService(gemini\_api\_key)

# Build entity context for Gemini

self.entity\_context = self.\_build\_entity\_context(prism\_data)

def \_build\_entity\_context(self, prism\_data: pd.DataFrame) -> Dict[str, List]:

"""Build context of known entities for Gemini"""

return {

'projects': list(prism\_data['project\_name'].dropna().unique()) + list(prism\_data['rev\_mapping'].dropna().unique()),

'edls': list(prism\_data['edl\_name'].dropna().unique()),

'locations': list(prism\_data['utilization\_location'].dropna().unique())

}

def parse\_query(self, user\_query: str) -> Dict[str, Any]:

"""Main parsing pipeline with Gemini LLM"""

try:

# Step 1: Extract entities using Gemini LLM

with st.spinner("🔍 Extracting entities with Gemini..."):

extracted\_entities = self.gemini\_service.extract\_entities\_with\_retry(

user\_query, self.entity\_context

)

# Step 2: Resolve entities with fuzzy matching

with st.spinner("🎯 Resolving entities with fuzzy matching..."):

resolved\_entities = self.\_resolve\_entities(extracted\_entities)

# Step 3: Route to appropriate agent using Gemini

with st.spinner("🚀 Determining best agent..."):

agent\_result = self.gemini\_service.route\_to\_agent\_with\_retry(

user\_query, extracted\_entities

)

# Step 4: Combine results

parsed\_result = self.\_format\_complete\_result(user\_query, agent\_result, resolved\_entities)

st.success("✅ Enterprise parsing complete!")

return parsed\_result

except Exception as e:

logger.error(f"Enterprise parsing failed: {e}")

st.error("⚠️ Enhanced parsing failed, using fallback")

return self.\_fallback\_parse(user\_query)

def \_resolve\_entities(self, entities: List[Dict]) -> List[Dict]:

"""Resolve entities using fuzzy matching"""

resolved\_entities = []

for entity in entities:

entity\_type = entity["type"]

raw\_value = entity["value"]

if entity\_type == EntityType.PROJECT.value:

resolution = self.entity\_resolver.resolve\_project(raw\_value)

elif entity\_type == EntityType.EDL.value:

resolution = self.entity\_resolver.resolve\_edl(raw\_value)

elif entity\_type == EntityType.LOCATION.value:

resolution = self.entity\_resolver.resolve\_location(raw\_value)

elif entity\_type == EntityType.TIME\_PERIOD.value:

resolution = self.time\_extractor.extract\_time\_period(raw\_value)

else:

resolution = {"resolved\_name": raw\_value, "confidence": 1.0, "method": "direct"}

resolved\_entity = {

\*\*entity,

"resolved\_value": resolution.get("resolved\_name"),

"confidence": resolution.get("confidence", 0.0),

"resolution\_method": resolution.get("method", "direct"),

"alternatives": resolution.get("alternatives", [])

}

resolved\_entities.append(resolved\_entity)

return resolved\_entities

def \_format\_complete\_result(self, user\_query: str, agent\_result: Dict, entities: List[Dict]) -> Dict[str, Any]:

"""Format complete parsing result"""

agent\_type\_map = {

"qa\_agent": AgentType.QA,

"insight\_agent": AgentType.INSIGHT,

"what\_if\_agent": AgentType.WHAT\_IF,

"recommendation\_agent": AgentType.RECOMMENDATION

}

return {

"original\_query": user\_query,

"agent\_type": agent\_type\_map[agent\_result["agent\_type"]],

"entities": entities,

"filters": self.\_build\_filters(entities),

"intent": self.\_determine\_intent(agent\_result["agent\_type"], entities),

"requires\_llm": agent\_result["agent\_type"] != "qa\_agent",

"confidence": agent\_result["confidence"],

"routing\_method": "gemini\_enhanced",

"llm\_reasoning": agent\_result.get("reasoning", ""),

"timestamp": datetime.datetime.now().isoformat()

}

def \_build\_filters(self, entities: List[Dict]) -> Dict[str, Any]:

"""Build filters from resolved entities"""

filters = {}

for entity in entities:

if entity["resolved\_value"] and entity["confidence"] > 0.7:

if entity["type"] == "project":

filters["project\_name"] = entity["resolved\_value"]

elif entity["type"] == "edl":

filters["edl\_name"] = entity["resolved\_value"]

elif entity["type"] == "location":

filters["location"] = entity["resolved\_value"]

elif entity["type"] == "time\_period":

filters["time\_period"] = entity["resolved\_value"]

return filters

def \_determine\_intent(self, agent\_type: str, entities: List[Dict]) -> str:

"""Determine detailed intent from agent type and entities"""

metrics = [e for e in entities if e["type"] == "metric"]

if agent\_type == "qa\_agent":

if metrics:

return f"Retrieve {', '.join([m['value'] for m in metrics])} metrics"

return "Factual data retrieval"

elif agent\_type == "insight\_agent":

return "Root cause and trend analysis"

elif agent\_type == "what\_if\_agent":

return "Scenario simulation and impact analysis"

elif agent\_type == "recommendation\_agent":

return "Action planning and optimization"

return "General inquiry"

def \_fallback\_parse(self, user\_query: str) -> Dict[str, Any]:

"""Fallback parsing when enhanced parsing fails"""

return {

"original\_query": user\_query,

"agent\_type": AgentType.QA,

"entities": [],

"filters": {},

"intent": "General factual inquiry",

"requires\_llm": False,

"confidence": 0.5,

"routing\_method": "fallback"

}

# Mock data for demonstration

def create\_mock\_prism\_data():

return pd.DataFrame({

'project\_name': ['Google Street View', 'Waymo Rider Exp', 'Google GLAD', 'Apple Dora'],

'rev\_mapping': ['Google Hardware Operations', 'Google Waymo', 'Google Ads', 'Apple Maps'],

'edl\_name': ['Ravikumar', 'John Smith', 'Maria Garcia', 'David Chen'],

'utilization\_location': ['India', 'USA', 'Philippines', 'India']

})

# Mock agent system for demonstration

class MockAgentSystem:

def process\_with\_agent(self, parsed\_query: Dict) -> str:

agent\_type = parsed\_query["agent\_type"].value

base\_response = f"""

\*\*Query:\*\* {parsed\_query['original\_query']}

\*\*Resolved Entities:\*\*

"""

for entity in parsed\_query.get('entities', []):

if entity.get('resolved\_value'):

base\_response += f"- {entity['type']}: {entity['resolved\_value']} ({(entity['confidence']\*100):.0f}% confidence)\n"

base\_response += f"\n\*\*Agent:\*\* {agent\_type.replace('\_', ' ').title()}\n"

base\_response += f"\*\*Intent:\*\* {parsed\_query['intent']}\n"

if parsed\_query.get('llm\_reasoning'):

base\_response += f"\*\*LLM Reasoning:\*\* {parsed\_query['llm\_reasoning']}\n"

base\_response += f"\n\*\*Routing Method:\*\* {parsed\_query['routing\_method']}\n"

base\_response += f"\*\*Overall Confidence:\*\* {parsed\_query['confidence']\*100:.0f}%\n\n"

base\_response += "\*Enterprise parsing completed. Ready for agent implementation.\*"

return base\_response

# Main Streamlit App

def main():

st.set\_page\_config(

page\_title="Workforce Analytics AI - Enterprise",

page\_icon="🚀",

layout="wide"

)

# Initialize session state

if 'chat\_history' not in st.session\_state:

st.session\_state.chat\_history = []

if 'prism\_data' not in st.session\_state:

st.session\_state.prism\_data = create\_mock\_prism\_data()

# Header

st.title("🚀 Enterprise Workforce Analytics AI")

st.markdown("\*\*Gemini-Powered • Fuzzy Matching • Enterprise-Grade Parsing\*\*")

# Sidebar with configuration

with st.sidebar:

st.header("⚙️ Configuration")

# Gemini API Key input

gemini\_api\_key = st.text\_input(

"Enter Gemini API Key:",

type="password",

help="Get your API key from Google AI Studio"

)

if gemini\_api\_key:

if 'query\_parser' not in st.session\_state:

with st.spinner("Initializing enterprise parser..."):

st.session\_state.query\_parser = EnterpriseQueryParser(

st.session\_state.prism\_data, gemini\_api\_key

)

st.success("✅ Enterprise parser initialized!")

st.header("🔍 Capabilities")

st.markdown("""

\*\*Gemini LLM Integration:\*\*

- Entity extraction

- Agent routing

- Intent understanding

\*\*Fuzzy Matching:\*\*

- Project name resolution

- EDL nickname support

- Multi-source matching

\*\*Enterprise Features:\*\*

- Retry logic & error handling

- Confidence scoring

- Fallback mechanisms

""")

if st.button("Clear Chat History"):

st.session\_state.chat\_history = []

# Main chat interface

col1, col2 = st.columns([3, 1])

with col1:

user\_query = st.chat\_input("Ask about utilization, NBL, projects, or optimization...")

with col2:

with st.expander("💡 Example Queries"):

st.write("• 'Show Ravi's project utilization'")

st.write("• 'Why is Google Street View NBL high?'")

st.write("• 'What if I reduce 5 NBL?'")

st.write("• 'How to achieve 85% utilization?'")

# Process user query

if user\_query:

# Add user message to chat

st.session\_state.chat\_history.append({"role": "user", "content": user\_query})

if 'query\_parser' not in st.session\_state:

st.error("❌ Please enter Gemini API key in sidebar first")

return

# Parse query with enterprise system

parsed\_query = st.session\_state.query\_parser.parse\_query(user\_query)

# Display detailed parsing results

with st.expander("🎯 Enterprise Parsing Details", expanded=True):

col1, col2, col3, col4 = st.columns(4)

with col1:

agent\_display = parsed\_query["agent\_type"].value.replace('\_agent', '').upper()

st.metric("🤖 Agent", agent\_display)

with col2:

st.metric("🎯 Confidence", f"{parsed\_query['confidence']\*100:.0f}%")

with col3:

method\_display = parsed\_query['routing\_method'].replace('\_', ' ').title()

st.metric("🔧 Method", method\_display)

with col4:

llm\_status = "Required" if parsed\_query["requires\_llm"] else "Not Needed"

st.metric("🧠 LLM", llm\_status)

# Intent and reasoning

st.info(f"\*\*🎯 Intent:\*\* {parsed\_query['intent']}")

if parsed\_query.get('llm\_reasoning'):

st.success(f"\*\*💭 LLM Reasoning:\*\* {parsed\_query['llm\_reasoning']}")

# Resolved entities section

if parsed\_query['entities']:

st.subheader("📋 Resolved Entities")

for entity in parsed\_query['entities']:

if entity.get('resolved\_value'):

col1, col2, col3 = st.columns([2, 3, 2])

with col1:

st.write(f"\*\*{entity['type'].title()}:\*\*")

with col2:

confidence\_color = "🟢" if entity['confidence'] > 0.8 else "🟡" if entity['confidence'] > 0.6 else "🔴"

st.write(f"{confidence\_color} `{entity['raw\_value']}` → `{entity['resolved\_value']}`")

with col3:

st.write(f"({entity['confidence']\*100:.0f}% via {entity['resolution\_method']})")

if entity.get('alternatives'):

with st.expander(f"See alternatives for {entity['raw\_value']}"):

st.write(", ".join(entity['alternatives']))

# Applied filters

if parsed\_query['filters']:

st.subheader("🎛️ Applied Filters")

st.json(parsed\_query['filters'])

# Process with appropriate agent

if 'agent\_system' not in st.session\_state:

st.session\_state.agent\_system = MockAgentSystem()

agent\_response = st.session\_state.agent\_system.process\_with\_agent(parsed\_query)

# Add assistant response to chat

st.session\_state.chat\_history.append({"role": "assistant", "content": agent\_response})

# Display chat history

for message in st.session\_state.chat\_history:

if message["role"] == "user":

st.chat\_message("user").write(message["content"])

else:

st.chat\_message("assistant").write(message["content"])

if \_\_name\_\_ == "\_\_main\_\_":

main()