Q&A: Deploying Scalable Backend AI Agents

This document provides a detailed overview of the technologies and strategies for deploying scalable backend AI agents, inspired by the practices of modern AI companies like Airtop.ai.

Q1: How do you design the overall architecture for a scalable backend agent system?

Answer:

A scalable backend agent system is typically composed of several key components, each designed to handle a specific part of the workflow. The architecture is often a distributed system to ensure high availability and scalability.

Core Components:

1. **API Gateway:** The single entry point for all client requests. It routes requests to the appropriate microservice and can handle tasks like authentication, rate limiting, and logging.
    - **Key Frameworks:** Kong, AWS API Gateway, or a custom solution using a reverse proxy like NGINX.
2. **Agent Orchestration Service:** This is the "brain" of the system. It receives tasks, breaks them down into smaller steps, and dispatches them to the appropriate agent workers.
    - **Key Frameworks:** Custom logic built with Python (FastAPI, Flask) or Node.js (Express). For complex workflows, you can use orchestration engines like Netflix Conductor or Camunda.
3. **LLM Serving Layer:** A dedicated service for hosting and serving the large language models (LLMs). This allows you to scale the LLM resources independently from the rest of the system.
    - **Key Frameworks:** vLLM, Hugging Face's Text Generation Inference (TGI), or custom solutions using FastAPI or gRPC.
4. **Browser Automation Farm:** A fleet of cloud-based browsers (e.g., Chrome) that the agents can control to perform web-based tasks.
    - **Key Frameworks:** Selenium Grid, or a custom solution using Playwright or Puppeteer with a browser orchestration tool like Moon.
5. **Message Queue:** A message broker to handle asynchronous communication between the services. This is crucial for decoupling the services and ensuring that tasks are not lost if a service fails.
    - **Key Frameworks:** RabbitMQ, Apache Kafka, or a cloud-based solution like AWS SQS or Google Cloud Pub/Sub.
6. **Database:** A database to store task information, results, user data, and agent configurations.
    - **Key Frameworks:** PostgreSQL for structured data, and a vector database like Pinecone or Weaviate for storing embeddings for

semantic search.

Example Workflow:

1. A user sends a request to the API Gateway (e.g., "Extract all the job titles from the careers page of example.com").
2. The API Gateway forwards the request to the Agent Orchestration Service.
3. The Orchestration Service breaks the task down: "1. Navigate to example.com/careers. 2. Extract the text of all job titles. 3. Return the list of job titles."
4. The Orchestration Service sends a message to the Browser Automation Farm to navigate to the URL and extract the raw HTML.
5. The HTML is then sent to the LLM Serving Layer with a prompt to extract the job titles.
6. The LLM returns the structured data (e.g., a JSON object).
7. The Orchestration Service stores the result in the database and returns it to the user via the API Gateway.

Q2: How do you deploy and serve the LLMs for the agents?

Answer:

Serving LLMs efficiently is one of the most critical parts of the system. The goal is to achieve high throughput (requests per second) and low latency.

Key Strategies:

1. **Use a Dedicated LLM Serving Framework:** These frameworks are highly optimized for serving LLMs and provide features like continuous batching, quantization, and optimized CUDA kernels.

   - **Key Frameworks:**
     - **vLLM:** An open-source library from UC Berkeley that is one of the fastest and most popular options.
     - **Hugging Face Text Generation Inference (TGI):** A production-ready solution for serving a wide range of open-source models.
     - **NVIDIA Triton Inference Server:** A more general-purpose inference server that can be used for LLMs and other types of models.

2. **Containerize the Serving Layer:** The LLM serving framework is packaged into a Docker container, which makes it easy to deploy and scale.

3. **Deploy on GPU-Powered Infrastructure:** LLMs require powerful GPUs to run efficiently. You can use cloud-based GPU instances (e.g., AWS P3/G4, Google Cloud A2/G2) or on-premise hardware.

4. **Use Model Quantization:** This technique reduces the precision of the model's weights (e.g., from 16-bit to 8-bit or 4-bit), which can a small trade-off in accuracy.

Example using vLLM with FastAPI:

```python
# main.py
from fastapi import FastAPI
from vllm import LLM, SamplingParams

# Initialize the LLM
llm = LLM(model="mistralai/Mistral-7B-Instruct-v0.1")
app = FastAPI()

@app.post("/generate")
async def generate(prompt: str):
    sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
    outputs = llm.generate(prompt, sampling_params)
    return {"text": outputs[0].outputs[0].text}
```

Deployment with Docker and Kubernetes:

1. **Dockerfile:**

```dockerfile
FROM python:3.9-slim
WORKDIR /app
RUN pip install fastapi uvicorn vllm
COPY main.py .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

2. **Kubernetes Deployment:** A Kubernetes deployment YAML file would be created to manage the Docker containers, specifying the number of replicas and the GPU resources required.

Q3: How do you manage and scale the browser automation tasks?

Answer:

Managing a fleet of browsers for automation requires a robust and scalable solution. The key is to run the browsers in a headless environment and orchestrate them effectively.

Key Strategies:

1. **Use a Browser Automation Framework:**
   - **Key Frameworks:**

    - **Playwright (Microsoft):** A modern and powerful framework that supports Chromium, Firefox, and WebKit. It has excellent features for handling modern web applications.
    - **Puppeteer (Google):** The original headless Chrome automation library. It is still a very popular and reliable choice.

2. **Run Browsers in Docker Containers:** Each browser instance is run in its own Docker container. This provides isolation and makes it easy to manage dependencies.

3. **Use a Browser Orchestration Tool:**

    - **Key Frameworks:**
        - **Selenium Grid:** The classic solution for running tests in parallel across multiple machines. It can be used with Playwright and Puppeteer with some custom configuration.
        - **Moon:** A modern, open-source browser orchestration tool that is specifically designed for running browsers in Kubernetes. It provides a simple and scalable way to manage a large number of browser sessions.

Example with Playwright and a custom worker service:

```python
# worker.py
import asyncio
from playwright.async_api import async_playwright
import pika

async def main():
    # Connect to RabbitMQ
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()
    channel.queue_declare(queue='tasks')

    async def callback(ch, method, properties, body):
        task = json.loads(body)
        async with async_playwright() as p:
            browser = await p.chromium.launch(headless=True)
            page = await browser.new_page()
            await page.goto(task['url'])
            html = await page.content()
            # Send the HTML to the next service for processing
            print("Task complete")
            await browser.close()

    channel.basic_consume(queue='tasks', on_message_callback=callback, auto_ack=True)
    channel.start_consuming()
```

```python
if __name__ == "__main__":
    asyncio.run(main())
```

Q4: How do you ensure the system is scalable and reliable?

Answer:

Scalability and reliability are achieved through a combination of architectural choices, infrastructure decisions, and operational best practices.

Key Strategies:

1. **Microservices Architecture:** As discussed in Q1, a microservices architecture allows you to scale each component independently.
2. **Container Orchestration:**
    - **Key Frameworks:**
        - **Kubernetes:** The industry standard for container orchestration. It provides features like auto-scaling, self-healing, and service discovery.
3. **Infrastructure as Code (IaC):**
    - **Key Frameworks:**
        - **Terraform:** The most popular IaC tool. It allows you to define your infrastructure in code, which makes it easy to create, update, and replicate your environment.
4. **CI/CD (Continuous Integration/Continuous Deployment):**
    - **Key Frameworks:**
        - **GitHub Actions:** A popular and easy-to-use CI/CD platform that is integrated with GitHub.
        - **Jenkins:** A powerful and flexible open-source CI/CD server.
5. **Monitoring and Logging:**
    - **Key Frameworks:**
        - **Prometheus:** An open-source monitoring system that is the de facto standard for Kubernetes monitoring.
        - **Grafana:** A tool for visualizing metrics from Prometheus and other data sources.
        - **ELK Stack (Elasticsearch, Logstash, Kibana):** A popular solution for centralized logging.

By implementing these strategies, you can build a system that is not only scalable and reliable but also easy to manage and maintain.

Q5: What are the best practices for web scraping to avoid getting blocked?

Answer:

Web scraping can be challenging as many websites have anti-bot measures. Here are some best practices to avoid getting blocked:

1. **Respect `robots.txt`:** This file, found at the root of a website (e.g., `example.com/robots.txt`), provides rules for bots. While not legally binding, respecting these rules is good practice and can prevent you from getting blocked.

2. **Use a Realistic User-Agent:** The User-Agent string in your HTTP headers identifies your client. By default, HTTP libraries have a distinct User-Agent (e.g., `python-requests/2.28.1`). It's best to use a common browser User-Agent.

   - **Python Example (with `requests`):**

     ```
     import requests

     headers = {
         'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K
     }
     response = requests.get('https://example.com', headers=headers)
     ```

3. **Rotate Proxies:** Using a pool of IP addresses can prevent a single IP from being blocked. You can use a proxy provider service that offers a rotating proxy.

4. **Implement Delays and Jitter:** Making requests too quickly can trigger rate limiters. Introduce random delays between your requests to mimic human behavior.

5. **Use a Headless Browser for JavaScript-heavy Sites:** For sites that rely heavily on JavaScript to render content, a simple HTTP request won't be enough. Use a headless browser like Playwright or Puppeteer to render the page before extracting the content.

Q6: When would you choose Puppeteer over Playwright, and vice-versa?

Answer:

Both Puppeteer and Playwright are excellent for browser automation, but they have some key differences:

- **Puppeteer:**

  - **Developed by Google:** It has strong support for Chrome and the Chrome DevTools Protocol.
  - **JavaScript/TypeScript First:** It is primarily a Node.js library, so it's a natural choice for teams that are already using JavaScript or TypeScript on the backend.
  - **Maturity:** It has been around longer than Playwright and has a very large community.
- **Playwright:**
  - **Developed by Microsoft:** It has a team of former Puppeteer developers.
  - **Cross-Browser Support:** This is the biggest advantage of Playwright. It supports Chromium, Firefox, and WebKit out of the box.
  - **Language Support:** It has official APIs for Python, Java, and .NET, in addition to Node.js. This makes it a great choice for teams that use a variety of languages.
  - **Auto-Waits:** Playwright has a more sophisticated auto-waiting mechanism, which can make scripts more reliable.

Conclusion:

- **Choose Puppeteer if:** Your team is primarily using Node.js and you only need to support Chrome.
- **Choose Playwright if:** You need to support multiple browsers, or if your team is using Python, Java, or .NET.

Q7: How do you choose the right LLM for a given task?

Answer:

Choosing the right LLM depends on a variety of factors, including the task, performance requirements, and cost.

Key Factors:

1. **Task Type:**
   - **Instruction-Following:** For tasks that require following instructions (e.g., "Extract the names of all the people mentioned in this article"), instruction-tuned models like Mistral-7B-Instruct or GPT-4 are a good choice.
   - **Creative Writing:** For creative tasks, you might want a model that is more "imaginative," like Claude.
   - **Code Generation:** For generating code, models that have been specifically trained on code, like Code Llama, are ideal.
2. **Performance:**
   - **Latency:** For real-time applications, you'll need a model with low latency. Smaller models are generally faster.

    - **Throughput:** For batch processing, you'll want a model with high throughput.
3. **Cost:**
    - **Open-Source vs. Proprietary:** Open-source models (e.g., Llama 2, Mistral) can be self-hosted, which can be more cost-effective in the long run, but requires more infrastructure management. Proprietary models (e.g., GPT-4, Claude) are easier to use but can be more expensive.
4. **Context Window:** The context window is the amount of text the model can consider at one time. For tasks that require processing long documents, you'll need a model with a large context window.

Example:

- **Task:** Build a chatbot to answer questions about a specific product.
- **Choice:** A smaller, instruction-tuned model like Mistral-7B-Instruct would be a good choice. It's fast, has a good context window, and can be fine-tuned on your product documentation for better accuracy.

Q8: How would you convert a project from TypeScript to Python?

Answer:

Converting a project from TypeScript to Python is a significant undertaking that requires careful planning and execution.

Steps:

1. **Analyze the TypeScript Codebase:**
    - **Identify Dependencies:** List all the npm packages used in the project and find their Python equivalents.
    - **Understand the Architecture:** Identify the main components of the application and how they interact.
    - **Analyze the Type Definitions:** TypeScript's static typing provides a lot of information about the data structures used in the application. This will be very helpful when writing the Python code.
2. **Choose the Python Tech Stack:**
    - **Web Framework:** If the TypeScript project uses a web framework like Express, you'll need to choose a Python equivalent like FastAPI or Flask.
    - **Async Handling:** If the TypeScript project makes heavy use of `async/await`, you'll want to use Python's `asyncio` library.
3. **Rewrite the Code:**
    - **Start with the Core Logic:** Begin by rewriting the core business logic of the application.
    - **Use Type Hinting:** Use Python's type hinting to replicate the static typing of TypeScript. This will make the code more readable and

easier to maintain.
    - **Write Tests:** Write unit tests for the new Python code to ensure that it is working correctly.
4. **Migrate the Data:** If the application has a database, you'll need to migrate the data to the new Python application.

Example (TypeScript to Python):

- **TypeScript (Express):**

```
import express from 'express';
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- **Python (FastAPI):**

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

Q9: Can you provide a Python example of an agent that uses a tool?

Answer:

A common pattern in agent design is to give the agent access to a set of "tools" that it can use to perform actions. Here is an example of a simple agent that can use a "search" tool.

Code:

```
import os
from openai import OpenAI

# A simple search tool
def search(query: str) -> str:
    """Searches for a query and returns the results."""
    # In a real application, this would call a search engine API
```

```python
    return f"Search results for '{query}': ..."

# Initialize the OpenAI client
client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

# The main agent loop
def agent(prompt: str):
    # Create the list of tools
    tools = [
        {
            "type": "function",
            "function": {
                "name": "search",
                "description": "Searches for a query.",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "query": {
                            "type": "string",
                            "description": "The query to search for.",
                        },
                    },
                    "required": ["query"],
                },
            },
        }
    ]

    # Create the list of messages
    messages = [{"role": "user", "content": prompt}]

    # First, send the prompt and tools to the model
    response = client.chat.completions.create(
        model="gpt-4",
        messages=messages,
        tools=tools,
        tool_choice="auto",
    )

    # Check if the model wants to call a tool
    if response.choices[0].message.tool_calls:
        tool_call = response.choices[0].message.tool_calls[0]
        function_name = tool_call.function.name
        function_args = json.loads(tool_call.function.arguments)

        # Call the tool
```

```python
        if function_name == "search":
            search_result = search(query=function_args.get("query"))

            # Send the tool result back to the model
            messages.append(response.choices[0].message)
            messages.append(
                {
                    "tool_call_id": tool_call.id,
                    "role": "tool",
                    "name": function_name,
                    "content": search_result,
                }
            )

            # Get the final response from the model
            final_response = client.chat.completions.create(
                model="gpt-4",
                messages=messages,
            )
            return final_response.choices[0].message.content
    else:
        return response.choices[0].message.content

# Run the agent
print(agent("What is the weather in San Francisco?"))
```

This example demonstrates the basic “ReAct” (Reasoning and Acting) loop, where the model can reason about which tool to use, use the tool, and then generate a final response based on the tool’s output.