TypeScript ile Bir AI Agent Oluşturma

Gerçek, çalışan bir agent loop inşa edin: deterministik, açıkça simüle edilmiş bir karar adımı, "Building an MCP Server"daki gerçek MCP tool'larını gerçekten çağırıyor.

Orta 30 dk

Son güncelleme: 5 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/building-an-ai-agent

---

TypeScript ile Bir AI Agent Oluşturma

Bu kategorideki her ders şu ana kadar kavramsaldı: "Bir AI Agent Nedir?", gözlemle-karar ver-eyleme geç loop'unu tanımladı, "Agent Planlama ve Akıl Yürütme Kalıpları", "karar ver" adımının nasıl çalışabileceğine daha derinlemesine indi, ve "Agent Davranışını Kontrol Etmek", böyle bir loop'un ihtiyaç duyduğu korkulukları kapsadı. Bu ders bir tane inşa ediyor -- "TypeScript ile MCP Sunucusu Oluşturma"daki tam olarak aynı `GeoFactsServer.ts` MCP server'ını yeniden kullanan, gerçek, çalışan bir agent loop -- ve aşağıdakilerin hepsi, tam olarak gösterildiği gibi davrandığını doğrulamak için gerçekten yazıldı, derlendi, ve çalıştırıldı.

> Bu ders gerçek bir LLM API'sini çağırmıyor. Devam etmeden önce "Bu Derste Ne Gerçek, Ne Simüle Edilmiş?"ı okuyun -- bunun tam olarak ne anlama geldiğini ve nedenini açıklıyor, böylece dersin geri kalanı olduğundan fazlasını iddia ediyormuş gibi yanlış okunmuyor.

Ne İnşa Edeceğiz?

1. "TypeScript ile MCP Sunucusu Oluşturma"daki `GeoFactsServer.ts`'yi değiştirmeden yeniden kullanacağız -- aynı `get_capital_city` ve `calculate_sum` tool'ları.
2. `AgentLoop.ts` ekleyeceğiz: bir karar adımı ve onun etrafında inşa edilmiş gerçek bir agent loop.
3. `RunAgentDemo.ts` ekleyeceğiz: gerçek bir MCP client'ını server'a bağlayan ve loop'u gerçek bir hedefe karşı çalıştıran bağlantı kodu.
4. Hepsini kuracak, derleyecek, çalıştıracak, ve ürettiği gerçek, adım adım izi inceleyeceğiz.

Bu Derste Ne Gerçek, Ne Simüle Edilmiş?

Bu dersin tüm meselesi, "Bir AI Agent Nedir?"deki agent loop'unu gerçek, çalışan kodla somutlaştırmaktır -- ama bunu dürüstçe yapmak, bir şey konusunda açık olmayı gerektiriyor: bu dersin gerçek bir LLM API'sine erişimi yok, dolayısıyla birini çağıramıyor. Aşağıdaki kod için bunun tam olarak ne anlama geldiği şu:

- **Gerçek:** agent loop'un kendisi -- "Agent Loop: Gözlemle, Karar Ver, Eyleme Geç"deki gözlemle-karar ver-eyleme geç döngüsü -- `AgentLoop.ts` içinde gerçekten, adım adım çalışıyor.
- **Gerçek:** her tool call. `AgentLoop.ts`, gerçek MCP `Client`'ının `callTool()`'unu çağırıyor, bu da ("Kavramlardan Wire Format'a: JSON-RPC 2.0"ın anlattığı gibi) gerçek bir `tools/call` mesajını gerçek `GeoFactsServer.ts` server'ına gönderiyor, o da gerçekten `get_capital_city`'yi ya da `calculate_sum`'ı çalıştırıp gerçek bir sonuç döndürüyor.
- **Gerçek:** adım sınırı -- loop, "Agent Davranışını Kontrol Etmek"teki "Adım ve İterasyon Sınırları"nın anlattığı gibi, gerçekten sınırlı.
- **Simüle edilmiş:** *karar* adımı. Gerçek bir agent, hedefi okumak ve açık uçlu akıl yürütme yoluyla sırada ne yapacağına karar vermek için bir dil modeli kullanır. Bu ders bunu `decideNextAction()` ile değiştiriyor: yalnızca iki sabit metin kalıbını ("capital of X" ve "sum of ...") tanıyan, küçük, tamamen deterministik, elle yazılmış bir fonksiyon. Bu, hem kod yorumlarında hem de yazdırdığı çıktıda bilinçli olarak `(simulated)` diye etiketleniyor -- bu bir dil modeli değildir, hedefi bir modelin anlayacağı şekilde "anlamaz," ve gerçek LLM davranışının bir gösterimi olarak okunmamalıdır.

Bu şekilde inşa edilmesinin nedeni: bu dersin amacı, "TypeScript ile MCP Sunucusu Oluşturma"daki gerçek MCP altyapısını kullanarak agent loop -> tool call -> MCP server -> tool result -> agent loop akışının gerçekten çalıştığını göstermektir -- gerçek bir LLM API entegrasyonunu göstermek değildir, ki bu dersin sonundaki "Gerçek Bir Modelle Ne Değişirdi?" bunu bunun yerine kavramsal olarak kapsıyor.

Proje Kurulumu (Building an MCP Server'ı Yeniden Kullanmak)

Bu ders, "TypeScript ile MCP Sunucusu Oluşturma"yla tam olarak aynı proje kurulumunu kullanır: aynı `package.json` (`"type": "module"` ve `@modelcontextprotocol/sdk` ile `zod`'a bağımlılıklarla) ve tam olarak aynı `tsconfig.json` (`"module": "NodeNext"`, `"moduleResolution": "NodeNext"`, `outDir: "dist"`). O proje klasörünüz hâlâ duruyorsa, bu dersin üç dosyasını doğrudan içine ekleyebilirsiniz. Yoksa, `package.json`'ı ve `tsconfig.json`'ı "TypeScript ile MCP Sunucusu Oluşturma"daki "package.json Dosyasını Oluşturun" ve "tsconfig.json Dosyasını Oluşturun"da gösterildiği gibi tam olarak yeniden oluşturun, sonra devam etmeden önce bir kez `npm install` çalıştırın.

```
ai-agent-demo/
├── package.json
├── tsconfig.json
├── GeoFactsServer.ts
├── AgentLoop.ts
└── RunAgentDemo.ts
```

Server'ı Yeniden Kullanmak: GeoFactsServer.ts

`GeoFactsServer.ts`, "TypeScript ile MCP Sunucusu Oluşturma"dan tamamen değiştirilmeden buraya kopyalandı -- aynı `get_capital_city` ve `calculate_sum` tool'ları, aynı `CAPITALS` sözlüğü (`France`, `Japan`, `Turkey`), bilinmeyen bir ülke için aynı `isError: true` davranışı:

```
// A small, self-contained MCP server exposing two tools: one that looks up
// information the model would not otherwise know (get_capital_city), and one
// that performs a computation (calculate_sum). Kept in its own file/function
// so it can be reused both by a real stdio-connected server process and,
// as in RunServerWithClient.ts, by an in-process demo client.
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { z } from "zod";

const CAPITALS: Record<string, string> = {
  France: "Paris",
  Japan: "Tokyo",
  Turkey: "Ankara",
};

export function createGeoFactsServer(): McpServer {
  const server = new McpServer({ name: "geo-facts-server", version: "1.0.0" });

  server.registerTool(
    "get_capital_city",
    {
      title: "Get Capital City",
      description: "Looks up the capital city of a given country.",
      inputSchema: { country: z.string().describe("Country name, e.g. 'France'") },
    },
    async ({ country }) => {
      const capital = CAPITALS[country];
      if (!capital) {
        return {
          content: [{ type: "text", text: `No capital known for "${country}".` }],
          isError: true,
        };
      }
      return { content: [{ type: "text", text: `The capital of ${country} is ${capital}.`
}] };
    }
  );

  server.registerTool(
    "calculate_sum",
    {
      title: "Calculate Sum",
      description: "Adds a list of numbers together.",
      inputSchema: { numbers: z.array(z.number()).describe("Numbers to add") },
    },
    async ({ numbers }) => {
      const total = numbers.reduce((acc, n) => acc + n, 0);
      return { content: [{ type: "text", text: `Sum: ${total}` }] };
    }
  );

  return server;
}
```

Bu dosyada agent'a özgü hiçbir şey yok. Bu bilinçli -- bir MCP server'ı, kendisini çağıran client'ın "TypeScript ile MCP Sunucusu Oluşturma"daki gibi tek bir tool call mı, yoksa bu dersteki gibi tam bir agent loop mu olduğunu bilmez. Tool tanımları, şemaları, ve

davranışları her iki durumda da tamamen değişmeden kalır -- tıpkı "Demo ile Gerçek Bir MCP Deployment Arasındaki Fark"ın transport'lar hakkında belirttiği gibi -- burada değişen server değil, *çağıran taraf*.

Agent Loop: AgentLoop.ts

`AgentLoop.ts`, "Bu Derste Ne Gerçek, Ne Simüle Edilmiş?"te anlatılan her iki yarıyı da içerir: simüle edilmiş `decideNextAction()` fonksiyonu, ve onun etrafında inşa edilmiş gerçek `runAgentLoop()` fonksiyonu.

```
// This file implements a genuine agent loop: a decision step chooses an
// action, that action is really executed against a real MCP server (through
// a real MCP client's callTool -- the exact same call RunServerWithClient.ts
// in "Building an MCP Server" used), and the real result is fed back into
// the next decision step. None of the tool-calling or MCP communication
// below is faked.
//
// What IS faked, on purpose, is the decision step itself. A real agent uses
// a large language model to read the goal and the conversation so far and
// decide, through open-ended reasoning, what to do next. This lesson has no
// access to a real LLM API, so decideNextAction() below is a small, fully
// deterministic, hand-written function: it recognizes two fixed patterns of
// text ("capital of X" and "sum of ...") and nothing else. It is NOT a
// language model, does not "understand" the goal in any general sense, and
// must never be mistaken for real model behavior -- it exists only so the
// *loop* around it (decide -> act -> observe -> repeat) can be demonstrated
// with a real, running, inspectable agent, calling the real
// get_capital_city and calculate_sum tools from GeoFactsServer.ts.
import type { Client } from "@modelcontextprotocol/sdk/client/index.js";

export interface AgentStep {
  thought: string;
  toolCall?: { name: string; arguments: Record<string, unknown> };
  toolResult?: string;
  toolIsError?: boolean;
}

export interface AgentResult {
  steps: AgentStep[];
  finalAnswer: string;
  stoppedByStepLimit: boolean;
}

interface Decision {
  thought: string;
  toolCall?: { name: string; arguments: Record<string, unknown> };
  finalAnswer?: string;
}

// (Simulated model.) A deterministic, explicitly hand-written stand-in for
// what a language model would do at each turn: look at the goal and what
// has happened so far, and decide on ONE next action. Real agents replace
// this single function with an LLM call -- runAgentLoop() below, and
// everything it does with the result, stays exactly the same either way.
function decideNextAction(goal: string, history: AgentStep[]): Decision {
```

```
  const alreadyCalled = (toolName: string) =>
    history.some((step) => step.toolCall?.name === toolName);

  const countryMatch = goal.match(/capital of (\w+)/i);
  if (countryMatch && !alreadyCalled("get_capital_city")) {
    const country = countryMatch[1];
    return {
      thought: `(simulated) Goal asks for the capital of "${country}". Plan: call
get_capital_city.`,
      toolCall: { name: "get_capital_city", arguments: { country } },
    };
  }

  const sumMatch = goal.match(/sum of ([0-9, and]+[0-9])/i);
  if (sumMatch && !alreadyCalled("calculate_sum")) {
    const numbers = sumMatch[1]
      .split(/,|and/i)
      .map((part) => part.trim())
      .filter((part) => part.length > 0)
      .map(Number);
    return {
      thought: `(simulated) Goal asks for the sum of [${numbers.join(", ")}]. Plan: call
calculate_sum.`,
      toolCall: { name: "calculate_sum", arguments: { numbers } },
    };
  }

  // Both sub-goals this demo cares about are already satisfied (or never
  // matched at all) -- (simulated) combine what the real tool calls
  // returned into a final answer instead of calling anything else.
  const facts = history
    .filter((step) => step.toolResult !== undefined)
    .map((step) => step.toolResult)
    .join(" ");
  return {
    thought: "(simulated) All sub-goals are covered by prior tool results. Plan: answer
using them.",
    finalAnswer: facts.length > 0 ? facts : "(no tool results were gathered to answer
from)",
  };
}

// The real agent loop: decide, act for real, observe the real result,
// repeat -- bounded by maxSteps so a decision function that never resolves
// to a final answer cannot run forever (see "Controlling Agent Behavior"
// for why a step-limit guardrail like this matters in general).
export async function runAgentLoop(
  client: Client,
  goal: string,
  maxSteps: number
): Promise<AgentResult> {
  const steps: AgentStep[] = [];

  for (let i = 0; i < maxSteps; i++) {
    const decision = decideNextAction(goal, steps);

    if (decision.finalAnswer !== undefined) {
      return { steps, finalAnswer: decision.finalAnswer, stoppedByStepLimit: false };
```

```
    }

    if (decision.toolCall) {
      const result = await client.callTool({
        name: decision.toolCall.name,
        arguments: decision.toolCall.arguments,
      });
      const content = result.content as Array<{ type: string; text: string }>;
      const text = content.map((c) => (c.type === "text" ? c.text : "")).join(" ").trim();
      steps.push({
        thought: decision.thought,
        toolCall: decision.toolCall,
        toolResult: text,
        toolIsError: Boolean(result.isError),
      });
    }
  }

  return {
    steps,
    finalAnswer: "(step limit reached before a final answer was produced)",
    stoppedByStepLimit: true,
  };
}
```

`decideNextAction()`, hedef metnine bakar ve iki regex'ten daha karmaşık hiçbir şey kullanmadan üç şeyden birine karar verir: hedef bir başkent belirtiyorsa ve o tool henüz çalışmadıysa `get_capital_city`'yi çağır, hedef bir toplam belirtiyorsa ve o tool henüz çalışmadıysa `calculate_sum`'ı çağır, ya da -- ikisi de geçerli değilse -- o ana kadar toplanan gerçek tool sonuçlarını bir nihai yanıtta birleştir. Ürettiği her `thought` string'i, bilinçli olarak, tam olarak `(simulated)` metniyle başlar, böylece bu demonun yazdırdığı hiçbir şey gerçek model akıl yürütmesiyle karıştırılamaz.

`runAgentLoop()`, "Agent Loop: Gözlemle, Karar Ver, Eyleme Geç"teki gerçek loop'tur: en fazla `maxSteps` iterasyonun her birinde, `decideNextAction()`'ı çağırır (gözlemle + karar ver), ve karar bir tool call içeriyorsa, gerçekten `client.callTool(...)`'u çalıştırır (eyleme geç) ve tekrar döngüye girmeden önce gerçek sonucu kaydeder. `maxSteps`'e bir nihai yanıt olmadan ulaşılırsa, loop sonsuza kadar devam etmek yerine `stoppedByStepLimit` `true` olarak döner -- bu, "Agent Davranışını Kontrol Etmek"teki "Adım ve İterasyon Sınırları"ndaki adım sınırı korkuluğudur, sıradan sınırlı bir `for` döngüsü olarak uygulanmıştır.

Agent'ı Çalıştırmak: RunAgentDemo.ts

`RunAgentDemo.ts`, gerçek bir MCP `Client`'ını, "TypeScript ile MCP Sunucusu Oluşturma"nın kullandığı aynı in-memory transport üzerinden `GeoFactsServer.ts`'ye bağlar, sonra `runAgentLoop()`'u gerçekten her iki tool'a da ihtiyaç duyan bir hedefe karşı çalıştırır:

```
// Wires a real MCP client and the real GeoFactsServer.ts server together
// over an in-memory transport (the same pattern RunServerWithClient.ts used
// in "Building an MCP Server"), then runs a real agent loop against them.
// The (simulated) decision-making in AgentLoop.ts is clearly labeled at
// every step below -- everything else (the tool calls, the MCP messages,
// the results) is genuinely executed.
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { InMemoryTransport } from "@modelcontextprotocol/sdk/inMemory.js";
import { createGeoFactsServer } from "./GeoFactsServer.js";
import { runAgentLoop } from "./AgentLoop.js";

async function main() {
  const server = createGeoFactsServer();
  const [clientTransport, serverTransport] = InMemoryTransport.createLinkedPair();
  const client = new Client({ name: "agent-demo-client", version: "1.0.0" });

  await Promise.all([
    server.connect(serverTransport),
    client.connect(clientTransport),
  ]);

  const goal = "What is the capital of Japan, and what is the sum of 12, 30, and 8?";
  console.log(`Goal: ${goal}\n`);

  const result = await runAgentLoop(client, goal, 5);

  result.steps.forEach((step, i) => {
    console.log(`Step ${i + 1}: ${step.thought}`);
    if (step.toolCall) {
      console.log(`  Tool call:   ${step.toolCall.name}(${JSON.stringify(step.toolCall.
arguments)})`);
      console.log(`  Tool result: ${step.toolResult}${step.toolIsError ? " (isError: true)"
: ""}`);
    }
    console.log();
  });

  console.log(`Final answer: ${result.finalAnswer}`);
  console.log(`Stopped by step limit: ${result.stoppedByStepLimit}`);

  await client.close();
  await server.close();
}

main();
```

Hedef -- `"What is the capital of Japan, and what is the sum of 12, 30, and 8?"` -- özellikle, doğru yanıtlamanın, agent loop'unun ilk sonuç geri geldikten sonra kendi kararıyla ikinci bir çağrı yapmasını gerektiren, sıra hâlinde iki ayrı gerçek tool call gerektirmesi için seçildi. `5` 'lik bir `maxSteps`, bu hedefin gerçekten ihtiyaç duyduğu iki adımın üzerinde rahat bir alan bırakırken, "Agent Davranışını Kontrol Etmek"teki adım sınırı korkuluğunu da yerinde tutuyor.

Bağımlılıkları Kurun, Derleyin, Çalıştırın

`package.json`, `tsconfig.json`, `GeoFactsServer.ts`, `AgentLoop.ts`, ve `RunAgentDemo.ts` hepsi aynı proje klasöründe kaydedilmişken, "TypeScript ile MCP Sunucusu Oluşturma"daki "Bağımlılıkları Kurun, Derleyin, Çalıştırın"ın kullandığı aynı üç komutu çalıştırın:

```
npm install
npx tsc -p tsconfig.json
node dist/RunAgentDemo.js
```

Bu üçüncü komut, tam olarak şunu üretir:

```
Goal: What is the capital of Japan, and what is the sum of 12, 30, and 8?

Step 1: (simulated) Goal asks for the capital of "Japan". Plan: call get_capital_city.
  Tool call:   get_capital_city({"country":"Japan"})
  Tool result: The capital of Japan is Tokyo.

Step 2: (simulated) Goal asks for the sum of [12, 30, 8]. Plan: call calculate_sum.
  Tool call:   calculate_sum({"numbers":[12,30,8]})
  Tool result: Sum: 50

Final answer: The capital of Japan is Tokyo. Sum: 50
Stopped by step limit: false
```

Burada Aslında Ne Oldu?

Adım 1: `decideNextAction()`, hedef metninde `"capital of Japan"` kalıbıyla eşleşti ve `get_capital_city`'yi çağırmaya (simulated) karar verdi. `runAgentLoop()` sonra gerçekten `client.callTool({ name: "get_capital_city", arguments: { country: "Japan" } })`'ı çağırdı -- tam olarak "MCP Client'ı Bağlayın: RunServerWithClient.ts"nin aynı tool için gösterdiği aynı MCP `tools/call` mesajı -- ve gerçek server `"The capital of Japan is Tokyo."` döndürdü.

Adım 2: `get_capital_city` artık zaten çağrılmış olarak işaretlenmişken, `decideNextAction()`, `"sum of 12, 30, and 8"` ile eşleşti, `[12, 30, 8]` sayılarını çıkardı, ve `calculate_sum`'ı çağırmaya (simulated) karar verdi. Gerçek tool call `"Sum: 50"` döndürdü.

Adım 3 (bir tool call olarak hiç çalışmasına gerek kalmadı): her iki alt-hedef de artık `history`'deki gerçek tool sonuçlarıyla kapsandığında, `decideNextAction()` ikisini birleştiren bir `finalAnswer` üretti, ve loop `maxSteps`'e ulaşmadan döndü. Çıktıdaki `stoppedByStepLimit: false`, loop'un alanı bittiği için değil, (simulated) karar adımının bittiğine karar verdiği için sonlandığını doğruluyor.

Hata Yolunu Denemek

Hedefi `"What is the capital of Wakanda?"` olarak değiştirip yeniden çalıştırmak şu gerçek çıktıyı üretir:

```
Goal: What is the capital of Wakanda?

Step 1: (simulated) Goal asks for the capital of "Wakanda". Plan: call get_capital_city.
  Tool call:   get_capital_city({"country":"Wakanda"})
  Tool result: No capital known for "Wakanda". (isError: true)

Final answer: No capital known for "Wakanda".
Stopped by step limit: false
```

Bu, "TypeScript ile MCP Sunucusu Oluşturma"daki "'Wakanda' Örneği Neden Var?"ın anlattığı aynı `isError: true` davranışının agent loop'una da doğru şekilde ulaştığını doğruluyor: `runAgentLoop()` çökmüyor ya da hatayı özel bir şekilde ele almıyor -- gerçek hata sonucu, diğer her tool sonucu gibi `history`'nin bir parçası oluyor, ve `decideNextAction()`'ın (simulated) nihai-yanıt adımı onu olduğu gibi bildiriyor.

Gerçek Bir Modelle Ne Değişirdi?

`decideNextAction()`'ı gerçek bir LLM call'uyla değiştirmek, bu agent'ın production versiyonunun ihtiyaç duyacağı tek değişikliktir -- `runAgentLoop()`, `GeoFactsServer.ts`, ve `RunAgentDemo.ts`'deki MCP bağlantı kodu tamamen aynı kalırdı, çünkü hiçbiri kararın nasıl verildiğine bağlı değil. Gerçek bir uygulama, modele hedefi, tool açıklamalarını ("Tools and Function Calling"deki "Bir Tool Tanımlamak: Ad, Açıklama ve Şema"nın anlattığı gibi name + description + schema), ve o ana kadarki geçmişi gönderir, ve sıradaki eylemi bir regex yerine modelin kendi akıl yürütmesinin -- "Agent Planlama ve Akıl Yürütme Kalıpları"ndaki kalıplardan birini izleyerek -- seçmesine izin verir. Bu dersin loop, gerçek tool call'lar, ve adım sınırı korkuluğu hakkında doğruladığı her şey, her iki durumda da değişmeden geçerli olur.

Best Practices

- Karar adımını ve loop mekaniğini, burada `decideNextAction()` ve `runAgentLoop()`'un yaptığı gibi ayrı fonksiyonlarda tutun -- bkz. "Gerçek Bir Modelle Ne Değişirdi?" -- bu, simüle edilmiş bir karar adımını gerçek bir model call'uyla değiştirmeyi izole, kapsanmış bir değişiklik hâline getiren şeydir.
- Simüle edilmiş ya da mock'lanmış bir bileşeni, hem kodda hem de ürettiği herhangi bir çıktıda açıkça etiketleyin -- bkz. "Bu Derste Ne Gerçek, Ne Simüle Edilmiş?" -- böylece kimse deterministik bir yer tutucuyu gerçek model davranışıyla karıştırmaz.

- Karar adımının kendi başına her zaman sonlanacağına güvenmek yerine, tam olarak `runAgentLoop()`'un yaptığı gibi, bir adım sınırını kodda zorunlu kılın -- bkz. "Agent Davranışını Kontrol Etmek"teki "Adım ve İterasyon Sınırları".

Yaygın Hatalar

- **Bu dersin `decideNextAction()`'ını basitleştirilmiş bir dil modeli sanmak.** "Bu Derste Ne Gerçek, Ne Simüle Edilmiş?"in açıkladığı gibi, yalnızca iki sabit metin kalıbını tanır, başka hiçbir şeyi tanımaz -- gerçek bir modelin genelliğine sahip değildir, ve kod ile çıktının ikisi de tam olarak bu yüzden `(simulated)` diye işaretler.
- **Gerçek bir LLM destekli agent'ın bu demodan farklı bir tool-calling mekanizmasına ihtiyacı olduğunu varsaymak.** "Gerçek Bir Modelle Ne Değişirdi?", `runAgentLoop()`'un gerçek tool-calling kodunun hiç değişmediğini gösterdi -- yalnızca karar adımı değişir.
- **Karar fonksiyonu "açıkça" sonlandığı için adım sınırını atlamak.** Buradaki `decideNextAction()` her zaman en fazla üç adımda sonlanıyor olabilir, ama "Agent Davranışını Kontrol Etmek"teki "Adım ve İterasyon Sınırları", sağlam bir sınırın loop'ta neden hâlâ yer aldığını kapsadı -- korkuluk, güvenilir şekilde sonlanmayan karar adımları için önemlidir, gerçek, model tarafından yönlendirilenler dahil.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bu dersin agent loop'u, tool call'ları, MCP iletişimi, ve adım sınırı hepsi gerçekten çalışan kod -- "TypeScript ile MCP Sunucusu Oluşturma"dan değiştirilmeden yeniden kullanılan `GeoFactsServer.ts` dahil.
- Yalnızca karar adımı, `AgentLoop.ts`'teki `decideNextAction()`, simüle edilmiş: gerçek model akıl yürütmesi için küçük, deterministik, açıkça etiketlenmiş bir yer tutucu, iki sabit metin kalıbını tanıyor.
- `runAgentLoop()`, "Agent Loop: Gözlemle, Karar Ver, Eyleme Geç"i sıradan sınırlı bir loop olarak uyguluyor, her tool eylemi için gerçek MCP `Client`'ının `callTool()`'unu çağırıyor ve gerçek sonucu kaydediyor.
- İki tool call gerektiren bir hedef (`get_capital_city` sonra `calculate_sum`), adım sınırına ulaşmadan doğru bir nihai yanıtla biten, doğrulanmış, gerçek, adım adım bir iz üretti.
- `decideNextAction()`'ı gerçek bir LLM call'uyla değiştirmek, bu dersin demosundan bir production agent'ına geçmek için gereken tek değişikliktir -- loop, tool call'lar, ve korkuluk hepsi aynı kalır.

Cheat Sheet

- Gerçek: agent loop, `client.callTool()` çağrıları, MCP mesajları, adım sınırı.

- Simüle edilmiş: `decideNextAction()` -- deterministik, regex tabanlı, göründüğü her yerde `(simulated)` diye etiketli.
- Loop şekli: `maxSteps`'e kadar `for` -> karar ver -> (tool call ise) gerçekten çalıştır + sonucu kaydet -> (nihai yanıt ise) erken dön.
- Dosyalar: `GeoFactsServer.ts` ("TypeScript ile MCP Sunucusu Oluşturma"dan değiştirilmemiş), `AgentLoop.ts` (karar + loop), `RunAgentDemo.ts` (bağlantı + 2 tool call gerektiren bir hedef).
- Çalıştırma sırası: `npm install` -> `npx tsc -p tsconfig.json` -> `node dist/RunAgentDemo.js`.

Terimler Sözlüğü

- **decideNextAction():** bu dersin simüle edilmiş karar adımı -- gerçek bir modelin akıl yürütmesinin yerini tutan, açıkça öyle etiketlenmiş deterministik bir fonksiyon.
- **runAgentLoop():** bu dersin gerçek agent loop'u -- geçmişi gözlemler, `decideNextAction()`'ı çağırır, gerçek tool call'lar çalıştırır, ve bir adım sınırını zorunlu kılar.
- **Adım sınırı (`maxSteps`):** "Agent Davranışını Kontrol Etmek"teki "Adım ve İterasyon Sınırları"ndan, `runAgentLoop()`'un çalıştıracağı karar ver-eyleme geç döngüsü sayısına sağlam, kodla zorunlu kılınan maksimum.
- **stoppedByStepLimit:** karar adımının nihai bir yanıta ulaştığı için biten bir loop'u, adım sınırı tarafından durmaya zorlanan bir loop'tan ayıran bu dersin bayrağı.