Week 6: Agent SDK - Running Agents Programmatically

01. Session Goals

- Understand the Claude Agent SDK architecture
- Use the `query()` function to run agents in TypeScript
- Build headless agents for automation
- Handle sessions, permissions, and error recovery

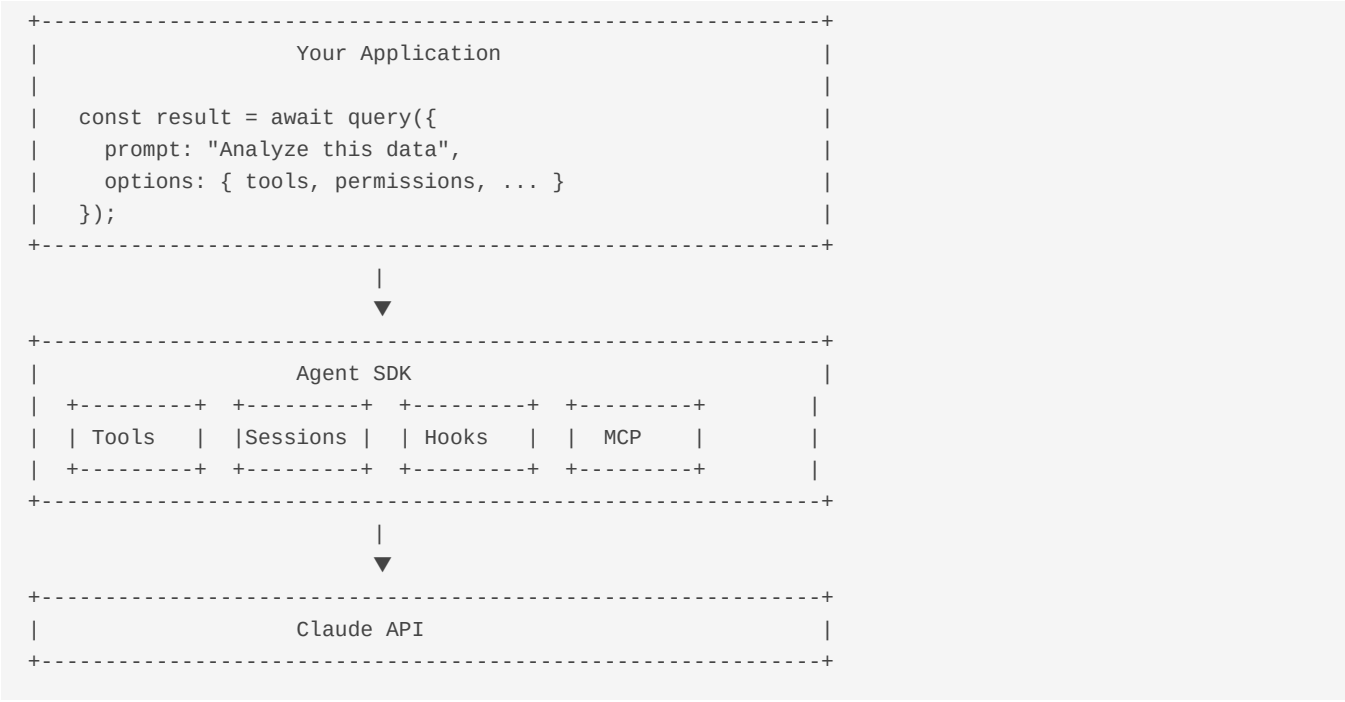
02. Block 1: Theory - The Agent SDK (30 min)

Why Run Agents Programmatically?
Claude Code is interactive. Great for developers. But for production:

- Need to run without human in the loop
- Need to process batches of tasks
- Need to integrate into existing systems
- Need to schedule and automate

The Agent SDK lets you embed Claude's agent capabilities in your code.

SDK Architecture



The `query()` Function

The core of the SDK. One function to run an agent:

```
import { query } from "@anthropic-ai/claude-agent-sdk";

const result = await query({
  prompt: "Research Acme Corp and summarize findings",
  options: {
    maxTurns: 10,
    systemPrompt: "You are a research analyst...",
  }
});

console.log(result.text);
```

Key Parameters

Parameter	Type	Description
`prompt`	string	The task for the agent
`options.maxTurns`	number	Maximum agentic loops
`options.systemPrompt`	string	Custom system instructions
`options.model`	string	Model to use (see below)
`options.tools`	Tool[]	Available tools
`options.mcpServers`	McpServer[]	MCP connections
`options.permissions`	Permission[]	Auto-granted permissions
`options.abortController`	AbortController	Cancellation signal

Model Selection

Choose the right model for your use case:

Model	ID	Best For	Cost
Sonnet 4.5	`claude-sonnet-4-5-20250514`	Most agent work (default)	$$
Opus 4.5	`claude-opus-4-5-20250514`	Complex reasoning	$$$$
Haiku	`claude-haiku-3-5-20241022`	Fast, simple tasks	$

```
const result = await query({
  prompt: "Complex analysis task",
  options: {
    model: "claude-opus-4-5-20250514",  // Use Opus for hard problems
    maxTurns: 10,
  }
});
```

Guidelines:

- Use Sonnet 4.5 for most production workloads (best balance of capability and cost)
- Use Opus 4.5 when you need stronger reasoning or the task is failing with Sonnet
- Use Haiku for sub-agents, simple lookups, or high-volume batch processing

Reference: Claude Models Overview ■

Built-in Tools

The SDK includes Claude Code's tools:

Tool	Purpose
`Read`	Read files
`Write`	Write files
`Edit`	Edit files
`Bash`	Execute commands
`Glob`	Find files by pattern
`Grep`	Search file contents
`WebSearch`	Search the web
`WebFetch`	Fetch web pages

Custom Tools

Beyond built-in tools, you can define your own. Custom tools in the Agent SDK are implemented as in-process MCP servers. This is how you connect Claude to any API, database, or service.

Creating a custom tool:

Custom tools require the `createSdkMcpServer` and `tool` helper functions. Use Zod for type-safe schemas:

```
import { query, tool, createSdkMcpServer } from "@anthropic-ai/claude-agent-sdk";
import { z } from "zod";

// Create an MCP server with custom tools
const crmServer = createSdkMcpServer({
  name: "crm-tools",
  version: "1.0.0",
  tools: [
    tool(
      "search_crm",
      "Search the CRM for contacts or companies by name",
      {
        searchQuery: z.string().describe("Search query (name, company, or email)"),
        recordType: z.enum(["contact", "company"]).describe("Type of record")
      },
      async (args) => {
        const response = await fetch(
          `https://api.crm.com/search?q=${args.searchQuery}&type=${args.recordType}`
        );
        const data = await response.json();

        return {
          content: [{
            type: "text",
            text: JSON.stringify(data, null, 2)
          }]
        };
      }
    )
  ]
});

// Use custom tools with streaming input (required for MCP)
async function* generateMessages() {
  yield {
    type: "user" as const,
    message: {
      role: "user" as const,
      content: "Find information about Acme Corp in our CRM"
    }
  };
}

for await (const message of query({
  prompt: generateMessages(),
  options: {
    mcpServers: {
      "crm-tools": crmServer
    },
    allowedTools: ["mcp__crm-tools__search_crm"]
  }
})) {
  if (message.type === "result" && message.subtype === "success") {
    console.log(message.result);
  }
}
```

Important notes:

- Custom MCP tools require streaming input mode (async generator for prompt)
- Tool names follow the pattern: `mcp__{server_name}__{tool_name}`
- Use Zod schemas for type-safe input validation

When to create custom tools:

Scenario	Example
Internal APIs	Company CRM, inventory system, billing
Third-party services	Stripe, Twilio, HubSpot
Database queries	Custom SQL against your warehouse
External data	Tavily, Firecrawl, Bright Data
Specialized logic	Lead scoring algorithm, pricing calculator

Custom tools vs external MCP servers:

External MCP servers (GitHub, Notion, PostgreSQL) are great for standardized, reusable integrations. In-process custom tools via `createSdkMcpServer` are better when you need:

- Tight integration with your application logic
- Custom authentication flows
- Business-specific operations
- Tools that only make sense for your use case

Demo
Live demo: Run a simple agent programmatically.

```
import { query } from "@anthropic-ai/claude-agent-sdk";

async function main() {
  const { text } = await query({
    prompt: "What files are in the current directory?",
    options: {
      maxTurns: 3,
    }
  });

  console.log(text);
}

main();
```

03. Block 2: Lab 1 - Your First SDK Agent (30 min)

Task: Build a File Analyzer Agent
Create an agent that analyzes CSV files and produces reports.

Step 1: Set up the project:

```
mkdir agents/file-analyzer
cd agents/file-analyzer
npm init -y
npm install @anthropic-ai/claude-agent-sdk typescript ts-node zod
npx tsc --init
```

Step 2: Create `src/index.ts`:

```
import { query } from "@anthropic-ai/claude-agent-sdk";

interface AnalysisResult {
  text: string;
  toolCalls: number;
}

async function analyzeFile(filePath: string): Promise<AnalysisResult> {
  let toolCalls = 0;

  const result = await query({
    prompt: `Analyze the CSV file at ${filePath}. Provide:
    1. Number of rows and columns
    2. Column names and data types
    3. Summary statistics for numeric columns
    4. Any data quality issues you notice`,
    options: {
      maxTurns: 5,
      onToolCall: (tool) => {
        toolCalls++;
        console.log(`Tool called: ${tool.name}`);
      }
    }
  });

  return {
    text: result.text,
    toolCalls
  };
}

// Run
analyzeFile('../data/sample-leads.csv')
  .then(result => {
    console.log('\n=== Analysis Result ===\n');
    console.log(result.text);
    console.log(`\nTotal tool calls: ${result.toolCalls}`);
  });
```

Step 3: Run the agent:

```
npx ts-node src/index.ts
```

Success Criteria

- [ ] Agent runs without errors
- [ ] File is analyzed correctly
- [ ] Tool calls are logged
- [ ] Output is structured

Discussion Questions

1. How many turns did the agent need?
2. Which tools did it use?

3. What would happen with a malformed CSV?

---

04. BREAK (10 min)

---

05. Block 3: Theory - Sessions and Streaming (30 min)

Sessions for Context Preservation
Sessions maintain state across multiple queries:

```
import { query, Session } from "@anthropic-ai/claude-agent-sdk";

// Create a session
const session = new Session({
  systemPrompt: "You are a data analyst helping with lead research.",
});

// First query - sets context
await query({
  prompt: "I'm working on analyzing leads for a B2B SaaS company.",
  session,
});

// Second query - has context from first
const result = await query({
  prompt: "What columns would be most important in a leads dataset?",
  session,
});

// Session remembers the conversation
```

When to Use Sessions

Scenario	Use Session?	Why
One-off task	No	No context needed
Multi-step workflow	Yes	Steps build on each other
Interactive conversation	Yes	Need conversation history
Batch processing (independent)	No	Each item is isolated
Batch processing (related)	Maybe	Depends on relationships

Streaming Responses

For long-running tasks, stream results:

```
import { query } from "@anthropic-ai/claude-agent-sdk";

async function streamingQuery() {
  for await (const message of query({
    prompt: "Research the top 5 CRM platforms and compare them",
    options: {
      maxTurns: 10,
    }
  })) {
    if (message.type === 'text') {
      process.stdout.write(message.content);
    } else if (message.type === 'tool_use') {
      console.log(`\n[Using tool: ${message.name}]\n`);
    } else if (message.type === 'tool_result') {
      console.log(`[Tool result received]\n`);
    }
  }
}
```

Message Types in Stream

Type	Description	Use
`text`	Text being generated	Show to user
`tool_use`	Tool being called	Log/display status
`tool_result`	Tool response	Debug/log
`error`	Error occurred	Handle gracefully
`done`	Query complete	Finalize

Permission Handling

Control what the agent can do:

```
const result = await query({
  prompt: "Update the config file with new settings",
  options: {
    permissions: {
      // Auto-approve these
      allow: [
        { tool: 'Read', pattern: '*' },
        { tool: 'Bash', pattern: 'npm:*' },
      ],
      // Auto-deny these
      deny: [
        { tool: 'Bash', pattern: 'rm:*' },
        { tool: 'Write', pattern: '*.env' },
      ],
      // Ask for these (default)
      // Everything else prompts the user
    },
    // Or run fully automated
    dangerouslyAllowAllTools: true, // Use with caution!
  }
});
```

Error Handling

Agents can fail. Handle gracefully:

```
import { query, AgentError, ToolError } from "@anthropic-ai/claude-agent-sdk";

try {
  const result = await query({
    prompt: "Risky operation...",
    options: { maxTurns: 5 }
  });
} catch (error) {
  if (error instanceof ToolError) {
    console.error(`Tool ${error.toolName} failed: ${error.message}`);
    // Maybe retry with different approach
  } else if (error instanceof AgentError) {
    console.error(`Agent error: ${error.message}`);
    // Log and alert
  } else {
    throw error; // Unknown error
  }
}
```

---

06. Block 4: Lab 2 - Build a Headless GTM Agent (45 min)

Task: Create a Lead Enrichment Service
Build a service that enriches leads automatically, running headlessly.

Step 1: Create project structure:

```
mkdir -p agents/lead-enricher/src
cd agents/lead-enricher
npm init -y
npm install @anthropic-ai/claude-agent-sdk typescript ts-node zod
```

Step 2: Create `src/types.ts`:

```
export interface Lead {
  name: string;
  company: string;
  email: string;
  title?: string;
}

export interface EnrichedLead extends Lead {
  companySize?: string;
  industry?: string;
  recentNews?: string[];
  linkedinUrl?: string;
  enrichedAt: Date;
  score?: number;
}

export interface EnrichmentResult {
  lead: EnrichedLead;
  success: boolean;
  error?: string;
  duration: number;
}
```

Step 3: Create `src/enricher.ts`:

```
import { query } from "@anthropic-ai/claude-agent-sdk";
import { Lead, EnrichedLead, EnrichmentResult } from './types';

const ENRICHMENT_PROMPT = `
You are a lead research specialist. Given a lead, research their company and provide:

1. Company size (employees)
2. Industry/vertical
3. Recent news (last 3 months)
4. LinkedIn profile URL if findable

Be concise. If you can't find information, say "Not found" for that field.

Output as JSON:
{
  "companySize": "...",
  "industry": "...",
  "recentNews": ["...", "..."],
  "linkedinUrl": "..."
}
`;

export async function enrichLead(lead: Lead): Promise<EnrichmentResult> {
  const startTime = Date.now();

  try {
    const result = await query({
      prompt: `Research this lead:
Name: ${lead.name}
Company: ${lead.company}
Email: ${lead.email}
Title: ${lead.title || 'Unknown'}

${ENRICHMENT_PROMPT}`,
      options: {
        maxTurns: 8,
        tools: ['WebSearch', 'WebFetch'],
        dangerouslyAllowAllTools: true,
      }
    });

    // Parse the JSON from response
    const jsonMatch = result.text.match(/\{[\s\S]*\}/);
    if (!jsonMatch) {
      throw new Error('No JSON found in response');
    }

    const enrichment = JSON.parse(jsonMatch[0]);

    const enrichedLead: EnrichedLead = {
      ...lead,
      ...enrichment,
      enrichedAt: new Date(),
    };

    return {
      lead: enrichedLead,
      success: true,
      duration: Date.now() - startTime,
    };
```

Step 4: Create `src/index.ts`:

```
import { enrichBatch } from './enricher';
import { Lead } from './types';
import * as fs from 'fs';

// Sample leads (in production, load from CSV or API)
const leads: Lead[] = [
  {
    name: "Sarah Chen",
    company: "TechCorp",
    email: "sarah@techcorp.com",
    title: "VP Engineering"
  },
  {
    name: "Mike Johnson",
    company: "DataFlow Inc",
    email: "mike@dataflow.io",
    title: "CTO"
  },
  // Add more leads...
];

async function main() {
  console.log(`Starting enrichment of ${leads.length} leads...\n`);

  const results = await enrichBatch(leads, 2);

  // Summary
  const successful = results.filter(r => r.success).length;
  const failed = results.filter(r => !r.success).length;
  const totalDuration = results.reduce((sum, r) => sum + r.duration, 0);

  console.log('\n=== Enrichment Complete ===');
  console.log(`Successful: ${successful}`);
  console.log(`Failed: ${failed}`);
  console.log(`Total time: ${(totalDuration / 1000).toFixed(1)}s`);
  console.log(`Avg per lead: ${(totalDuration / results.length / 1000).toFixed(1)}s`);

  // Save results
  fs.writeFileSync(
    'output/enriched-leads.json',
    JSON.stringify(results, null, 2)
  );
  console.log('\nResults saved to output/enriched-leads.json');
}

main().catch(console.error);
```

Step 5: Run the enricher:

```
mkdir -p output
npx ts-node src/index.ts
```

Bonus: Add Scoring

Extend the enricher to score leads after enrichment:

```
// In enricher.ts, add:

export async function scoreLead(lead: EnrichedLead): Promise<number> {
  const result = await query({
    prompt: `Score this lead from 0-100 based on:
- Company size (larger = higher score for B2B)
- Title seniority (VP/C-level = higher)
- Industry fit (tech = higher)

Lead:
${JSON.stringify(lead, null, 2)}

Return only a number 0-100.`,
    options: {
      maxTurns: 1,
    }
  });

  return parseInt(result.text.trim()) || 50;
}
```

Deliverable

- Working lead enrichment service
- Batch processing with concurrency control
- JSON output file with enriched leads
- Summary statistics

---

07. Wrap-Up (15 min)

Key Takeaways

1. `query()` is the core - One function to run agents programmatically
2. Sessions preserve context - Use for multi-step workflows
3. Stream for UX - Show progress on long tasks
4. Permissions matter - Control what agents can do in production
5. Handle errors - Agents fail; build resilient systems

Homework

Build an Automated Report Generator:

1. Create an agent that:
   - Reads a data file (CSV or JSON)
   - Analyzes the data
   - Generates a markdown report
   - Saves to output folder

2. Add features:
   - Streaming progress output
   - Error handling with retries
   - Session-based follow-up questions
3. Run it on 3 different datasets
4. Document:
   - Code structure
   - How you handled errors
   - Performance observations

Next Week Preview
Week 7: Evals - Building dashboards to track agent performance and iterate until alignment

---

08. Facilitator Notes

Common Issues
1. API key not set: Ensure `ANTHROPIC_API_KEY` environment variable
2. Rate limiting: Add delays between batch items
3. Tool permission errors: Check permission configuration
4. JSON parsing fails: Agent output isn't always clean JSON

Environment Setup
Participants need:

- Node.js 18+
- npm or yarn
- Anthropic API key
- TypeScript knowledge (basic)

Timing Adjustments
- Lab 1 is simpler, can extend discussion if concepts unclear
- Lab 2 is substantial, can simplify to single-lead enrichment
- Bonus scoring can be homework if time short

Code Quality Tips
- Emphasize type safety with TypeScript
- Show proper async/await patterns
- Discuss error handling strategies
- Mention rate limiting for production