



Lab 4: Spec Driven Development

Overview

In this lab, you will learn how to use Kiro's **Spec-Driven Development** feature to build complex applications through a structured, methodical approach. Unlike Vibe Coding's conversational style, Spec-Driven Development follows a formal software development lifecycle with clear phases and documentation.

What is Spec-Driven Development? Specs (specifications) are structured artifacts that formalize the development process for complex features in your application. They provide a systematic approach to transform high-level ideas into detailed implementation plans with clear tracking and accountability.

With Kiro's Spec-Driven Development, you can:

- **Break down requirements** into user stories with acceptance criteria using EARS notation
- **Build comprehensive design docs** with sequence diagrams and architecture plans
- **Track implementation progress** across discrete, manageable tasks
- **Collaborate effectively** between product and engineering teams
- **Maintain documentation** throughout the development process

What You'll Build:

- **AI-powered chatbot service** integrated with your existing e-commerce website
- **Python-based backend** using Strands Agents SDK and Nova Pro LLM
- **Independent microservice** with API integration to the main backend
- **Intelligent product recommendations** based on inventory data
- **Interactive chat interface** with action capabilities (add to cart, view products)

What You'll Learn:

- How to structure complex development projects using specifications
- Understanding the three-phase workflow: Requirements → Design → Implementation
- Working with AI agent frameworks and LLM integration
- Building microservices that integrate with existing applications
- Testing and validating AI-powered features

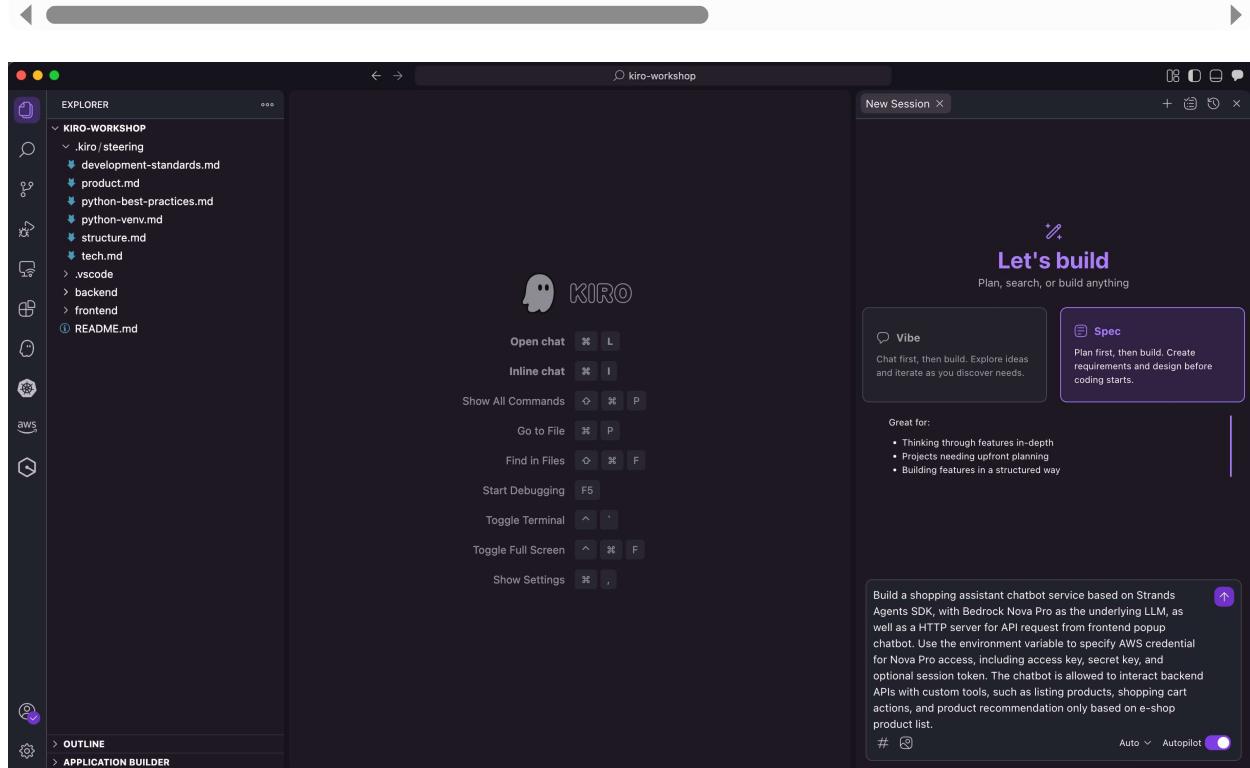
This lab demonstrates how Spec-Driven Development provides structure and accountability for complex feature development, ensuring thorough planning before implementation while maintaining clear progress tracking throughout the process.

Let's Build Chatbot with Spec-Driven Development

In Kiro's chat panel, open a new session. Choose **Spec** and send the prompt below:

Build a shopping assistant chatbot service based on Strands Agents SDK, with Bedrock Nova Pro as the underlying LLM, as well as a HTTP server for API request from frontend popup chatbot. Use the environment variable to specify AWS credential for Nova Pro access, including access key, secret key, and optional session token. The chatbot is allowed to interact backend APIs with custom tools, such as listing products, shopping cart actions, and product recommendation only based on e-shop product list.

Setup a HTTP server for API request from frontend popup chatbot component to the chatbot service. The chatbot is allowed to interact backend APIs with custom tools, such as listing products, shopping cart actions, and product recommendation only based on e-shop product list.



Kiro generates three key files that follow the three phase workflow through Requirements → Design → Implementation, to form the foundation of each specification:

1. requirements.md - Captures user stories and acceptance criteria in structured EARS notation
2. design.md - Documents technical architecture, sequence diagrams, and implementation considerations
3. tasks.md - Provides a detailed implementation plan with discrete, trackable tasks

1. Requirements

Simulate the SDLC Planning phase - The Development team collects requirements from several stakeholders such as customers, internal and external experts, and managers to create a software requirement specification document.

Review the requirements specification and modify if needed. Then click **Move to design phase**.

The screenshot shows the Kiro workshop interface with the 'Requirements' tab selected. On the left, a code editor displays a file named 'requirements.md' containing requirements for a shopping assistant chatbot. The code includes sections for Introduction, Glossary, Requirements, and specific requirements like 'Requirement 1'. On the right, a sidebar titled 'New Session' shows a list of tools and their descriptions, such as 'Called MCP tool search_docs' and 'Called MCP tool fetch_doc'. A message from 'Kiro' suggests creating a spec for the shopping assistant chatbot feature.

```

1 # Requirements Document
2
3 ## Introduction
4
5 This document specifies the requirements for a shopping assistant chatbot service built using the Strands Agents SDK with Amazon Bedrock Nova Pro as the underlying large language model. The chatbot service provides an HTTP API that enables frontend applications to interact with an AI-powered shopping assistant capable of accessing backend e-commerce APIs through custom tools. The system allows users to browse products, manage their shopping cart, and receive personalized product recommendations based on the available product catalog.
6
7 ## Glossary
8
9 - **Chatbot Service**: The HTTP server application that hosts the Strands Agent and exposes API endpoints for frontend interaction
10 - **Strands Agent**: The AI agent instance created using the Strands Agents SDK that processes user queries and executes tools
11 - **Bedrock Nova Pro**: Amazon Bedrock's Nova Pro foundation model used as the underlying LLM for the agent
12 - **Backend API**: The existing e-commerce REST API that provides product, cart, and review operations
13 - **Custom Tool**: A Python function decorated with @tool that enables the agent to interact with backend APIs
14 - **Frontend Client**: The React-based web application that sends user messages to the chatbot service
15 - **AWS Credentials**: Access key, secret key, and optional session token required for Bedrock API authentication
16
17 ## Requirements
18
19 ### Requirement 1
20
21 **User Story:** As a frontend developer, I want to send user messages to the chatbot service via HTTP POST requests, so that I can integrate the AI assistant into the web application.
22
23 #### Acceptance Criteria
24
25 1. THE Chatbot Service SHALL expose an HTTP endpoint at POST /api/chat that accepts JSON payloads containing user messages
26 2. WHEN a POST request is received at /api/chat, THE Chatbot Service SHALL validate that the request body contains a "message" field
27 3. IF the request body is missing the "message" field, THEN THE Chatbot Service SHALL return an HTTP 400 status code with an error message
28 4. WHEN a valid message is received, THE Chatbot Service SHALL invoke the Strands Agent with the user message
29 5. THE Chatbot Service SHALL return the agent's response as JSON with an HTTP 200 status code
30
31 #### Requirement 2

```

2. Design

Simulate the SDLC Design phase - Software Engineers analyze requirements and identify the best solutions to create the software. For example, they may consider integrating pre-existing modules, make technology choices, and identify development tools.

Review the design specification and modify if needed (eg. Nova Pro vs Nova Premier). Then click **Move to implementation plan**.

The screenshot shows the Kiro workshop interface with the 'Design' tab selected. On the left, a code editor displays a file named 'design.md' containing a design document for the shopping assistant chatbot. It includes sections for Overview, Architecture, and System Components, with a detailed diagram showing the flow from Frontend (React) to Chatbot Service (Python + Flask) via HTTP POST requests. On the right, the 'New Session' sidebar shows tool descriptions and a message from 'Kiro' suggesting creating a design document. A message from 'Great! Now I'll create the design document based on the approved requirements.' is also present.

```

1 # Design Document
2
3 ## Overview
4
5 The shopping assistant chatbot service is a Python-based HTTP server that integrates the Strands Agents SDK with Amazon Bedrock Nova Pro to provide an AI-powered conversational interface for e-commerce operations. The service acts as a middleware layer between the frontend React application and the existing Node.js backend API, enabling users to interact with the e-commerce platform through natural language.
6
7 The architecture follows a three-tier pattern:
8 1. Frontend (React) – User interface with chatbot popup
9 2. Chatbot Service (Python + Strands SDK) – AI agent with custom tools
10 3. Backend API (Node.js + Express) – E-commerce data and operations
11
12 ## Architecture
13
14 ### System Components
15
16 ...
17
18 [Frontend (React) Port 3000]
19 | HTTP POST /api/chat
20 | { "message": "..."}
21 |
22 [Chatbot Service (Python + Flask) Port 3002]
23 | Strands Agent
24 | - Bedrock Nova Pro
25 | - Custom Tools
26 |
27 | HTTP Requests (GET/POST/PUT/DELETE)
28
29
30
31
32
33
34
35
36

```

3. Implementation

Simulate the SDLC Implement phase - In the implementation phase, the development team codes the product. They analyze the requirements to identify smaller coding tasks they can do daily to achieve the final result.

Review the design specification and modify if needed. When task specification is generated, some tasks are marked as optional (e.g. unit tests, documentation) to focus on core features first. You can either **keep optional tasks or make all tasks required**.

In this workshop, choose **Keep optional tasks (faster MVP)** for efficiency.

The screenshot shows the Kiro workshop interface. On the left, the tasks.md file is open, displaying a hierarchical list of tasks under the '# Implementation Plan' section. Tasks include setting up the chatbot service project structure, implementing custom tools for backend API interaction, and creating tools module with list_products tool. On the right, a sidebar titled 'New Session X' shows a log of actions: 'Called MCP tool search_docs', 'Called MCP tool fetch_doc', and 'Created requirements.md'. Below this, a message says 'Now let me create the requirements document for the shopping assistant chatbot feature:' followed by a 'Created requirements.md' entry. A note says 'Do the requirements look good? If so, we can move on to the design.' Another message says 'Great! Now I'll create the design document based on the approved requirements.' A third 'Created design.md' entry is shown. A note says 'Does the design look good? If so, we can move on to the implementation plan.' A fourth 'Created tasks.md' entry is shown. A note says 'Perfect! Now I'll create the implementation task list based on the approved design.' A message at the bottom says 'The current task list marks some tasks (e.g. unit tests, documentation) as optional to focus on core features first.' A button labeled 'Keep optional tasks (faster MVP)' is highlighted. At the bottom right, there's a 'Revert' button and a message 'Ask a question or describe a task...'. A small icon for 'Auto' and 'Autopilot' is also visible.

```

# Implementation Plan
1. Start task
  1.1 Set up chatbot service project structure
    - Create 'chatbot/' directory in workspace root
    - Create Python virtual environment in 'chatbot/' directory
    - Create 'requirements.txt' with dependencies: strands-agents, flask, flask-cors, boto3, requests, python-dotenv
    - Create '.env.example' file documenting required environment variables
    - Requirements: 2.1, 2.2, 10.1, 10.2

  1.2 Start task
    1.2.1 Implement custom tools for backend API interaction
      - Requirements: 3.1, 3.2, 3.3, 4.1, 4.2, 5.1, 5.2, 6.1, 6.2, 7.1, 7.2, 8.1, 8.2

  1.2.2 Start task
    1.2.2.1 Create tools module with list_products tool
      - Create 'chatbot/tools.py' file
      - Implement 'list_products()' function with @tool decorator
      - Add HTTP GET request to 'http://localhost:3001/api/products'
      - Handle response and error cases
      - Requirements: 3.1, 3.2, 3.3

  1.2.2.2 Start task
    1.2.2.2.1 Implement add_to_cart tool
      - Add 'add_to_cart(product_id: int, quantity: int)' function to 'tools.py'
      - Add HTTP POST request to 'http://localhost:3001/api/cart'
      - Return success confirmation or error message
      - Requirements: 4.1, 4.2, 4.3, 4.4

  1.2.2.2.2 Start task
    1.2.2.2.2.1 Implement get_cart tool
      - Add 'get_cart()' function to 'tools.py'
      - Add HTTP GET request to 'http://localhost:3001/api/cart'
      - Return formatted cart contents
      - Requirements: 5.1, 5.2, 5.3

  1.2.2.2.2.2 Start task
    1.2.2.2.2.2.1 Implement update_cart_item tool

```

Task Execution

In task.md, click **⚡ Start task** button above each tasks one by one to execute the task, making necessary code changes and running commands. Or you can [execute all the tasks ↗](#) in your tasks.md file by send the following prompt to Kiro:

Execute all tasks in the spec



Note: we do not recommend to execute all tasks at once as we recommend a task-wise execution to get better results.

It takes around 20-30 minutes to complete all tasks. Once the Task Run are done, update AWS Credentials in .env file under chatbot service folder. If .env file doesn't exist, copy the .env.example file and rename to .env file for python-dotenv library to manage environment variable.

If AWS hosted event, go to [Workshop Studio Event Homepage ↗](#) and obtain the AWS Credential in **Get AWS CLI credentials**. This will open a popup with your temporary AWS credentials that you can copy the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY and AWS_SESSION_TOKEN with value to .env file. Keep AWS_REGION or AWS_DEFAULT_REGION to us-east-1. ⚠ The temporary AWS credentials expire after 1 hour. You will need to obtain again and update the value in .env after expiry.

For example:

```
AWS_ACCESS_KEY_ID=xxxxxxxx
AWS_SECRET_ACCESS_KEY=yyyyyyyy
```

```
AWS_SESSION_TOKEN=zzzzzzzzzzzzzzzzzzzzzz  
AWS_REGION=us-east-1
```

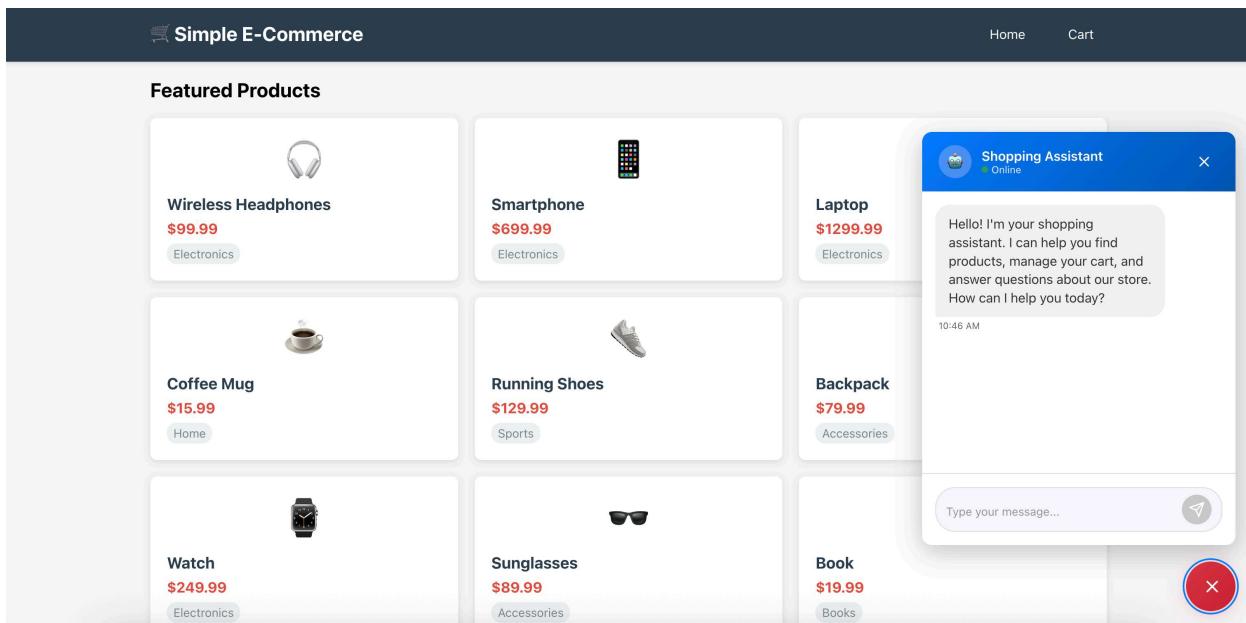
```
$ .env.example      .env      x
chatbot > .env
1 # AWS Credentials for Bedrock Nova Pro
2 AWS_ACCESS_KEY_ID=ASIA
3 AWS_SECRET_ACCESS_KEY=qnzh+
4 AWS_SESSION_TOKEN=IQoJb3
5 AWS_REGION=us-east-1
6
```

Send the prompt below to start the E-commerce website with AI Agent Chatbot:

Start the chatbot service, backend and frontend if not running.



The E-commerce website should be opened via a web browser automatically. If not, open any web browser, browse <http://localhost:3000> or other port for React Frontend. You will access the E-commerce website similar to this:



Testing the Chatbot

Once your e-commerce website is running, test the chatbot with the following questions:

1. Hello
2. What products do you have?
3. Add <product_name> to cart
4. Show me the cart
5. Clear my cart
6. What do you suggest to buy for hiking?
7. Add all suggested items to cart

If not work, describe the problem as prompt and send to Kiro for debug. Here are [some common issues](#) for reference.

The screenshot shows a grid of featured products: Wireless Headphones (\$99.99, Electronics), Smartphone (\$699.99, Electronics), Laptop (\$1299.99, Electronics), Coffee Mug (\$15.99, Home), Running Shoes (\$129.99, Sports), Backpack (\$79.99, Accessories), Watch (\$249.99, Electronics), Sunglasses (\$89.99, Accessories), and Book (\$19.99, Books). A floating 'Shopping Assistant' window is open, showing a message input field, a list of available products in the store, and a message history. The history includes a query about available products, a response listing the Electronics category with details for the Wireless Headphones and Smartphone, and a message from the user asking to add a Smartphone to the cart.

Troubleshooting Common Issues

Chatbot icon not appear

If chatbot icon doesn't appear in frontend, send this prompt to Kiro:

Chatbot icon doesn't appear in frontend.



No response / dummy response in Chatbot

If you encounter problems with the functionality, for example no response from the chatbot, send this prompt to Kiro:

Chatbot has no response or provide dummy response after I ask question in the frontend.



Wrong action from Chatbot

If you encounter problems with chatbot action, for example replying wrong product list or no action to shopping cart, send this prompt to Kiro:

Chatbot list the products out of the inventory in this e-commerce website.



Chatbot doesn't add the product to shopping cart from my request.



No update in Shopping Cart page

If you encounter problems in frontend rendering, for example no re-rendering in shopping cart page after chatbot add a product to cart (try to refresh browser to see if rendering problem), send this prompt to Kiro:

Shopping cart page doesn't update automatically after chatbot helps to add a product in cart.



Application Won't Start

If the application fails to start, try:

The application is not starting properly.



Vibe vs Spec Tips

When to Vibe

- Interactive Q&A Format:** Vibe sessions are optimized for back-and-forth conversations about code, allowing you to ask questions and get immediate responses.
- Quick Assistance:** They're ideal for getting quick answers to coding questions, explanations of code behavior, or understanding concepts without going through a formal specification process.
- Contextual Understanding:** Like other Kiro sessions, Vibe sessions leverage context providers to understand your codebase, but with a focus on explanation rather than extensive code generation.
- Flexible Approach:** Vibe sessions offer a more fluid, less structured approach compared to Spec sessions, making them suitable for exploratory coding and learning.

When to Spec

- Complex Development Tasks:** Use Spec sessions for building complex features, entire applications, or significant refactoring that requires careful planning and execution.
- Structured Approach:** When you need a methodical, step-by-step approach to development with clear documentation of requirements and implementation details.
- Team Collaboration:** For projects where multiple team members need to understand the implementation plan and track progress against specifications.
- Documentation Needs:** When you want to generate detailed documentation alongside your code implementation for future reference or knowledge sharing.

Previous

Next