International Journal of Latest Engineering and Management Research (IJLEMR)
ISSN: 2455-4847
www.ijlemr.com || Volume 10 – Issue 03 || March 2025 || PP. 37-47

Building Scalable Design Systems for Enterprise Applications: Integrating Figma and UI Components with Examples in React

Artsiom Marozau[1], Volha Kapitonava[2]
[1]*Software Engineering Team Lead, EPAM Systems, Poland*
[2]*UX Designer, IBM, Poland*

Abstract: The article describes the implementation methodology of reusable components across web applications and how to effectively use design tokens to achieve design consistency for large enterprise companies. It focused on the implementation of the front-end common components with react and integrating design tokens for UI consistency. Test automation and unit testing are also mentioned for better stability. The approach ensures that the system would be able to handle amounts of UI changes that can be isolated on the components level and wouldn't affect the business logic of the application where components are used.

Keywords: Design System, Figma, front-end, react, tests

1. Introduction

Problem statement: Large-scale enterprise projects often face challenges in maintaining UI components and keeping the UI consistent. While companies grow and develop multiple products constantly to ensure a cohesive and branded user experience it becomes a complex problem for the teams. Enterprise organizations aim to establish a unique visual identity through a unified design system — covering brand colors, component styles, behavior that help them to be different from competitors on the market. A consistent UI not only strengthens brand recognition across products but also improves user experience and delight helping users quickly adapt to new features and interfaces. However, without a centralized and well-maintained design system, inconsistencies can lead to a time increase for feature implementation and a reduction in user satisfaction which can lead to significant business losses [1].

Objective: Propose a clear and simple method to create a scalable design system that connects design and development teams. The focus will be on making components reusable, easy to automate, and consistent across all products.

Scope: Building components in Figma, with a focus on implementing them in JavaScript, integrating design tokens, and supporting themes, and creating a centralized showcase for component visualization and interaction, covering UI features with automated effective tests.

2. Approach to Building Enterprise Design Systems

2.1 Steps required to ensure system design consistency

Components structure: need to design a reusable set of components and components variants (i.e. set of Buttons, Inputs, DropDowns, Tables, Inlines messages etc.) that will cover all user interactions and can be easily reused across the application. The components structure can be copied from existing design systems like Material UI [2], this will save time on researching what components should be made for application.

2.2 Implementing components

2.2.1 How to make components

Here is the comparison table for 2 options on how to make the common components, let's see comparison table 1 to compare these options:

	Headless components	Web-components
Description	Headless components are components that contain only behavior logic and don't have any UI logic. For this case, I need to write styles according to the design system.	Web Components is a suite of different technologies allowing you to create reusable custom elements — with their functionality encapsulated away from the rest of your code — and utilize them in your web apps. [3]

Plus	- you already have components and their logic - a community that supports components and bug fixing in components - you can style components as you wish and no need to rewrite styles, no risks of regression with styles and focus on the design system - already help you to implement better support for A11Y	- fully isolated, so no one will affect the styles in component, so they look the same on every page - full control of changes and their delivery to end user - - framework agnostic that adds more flexibility
Minus	- a limited amount of components - code is owned by someone else and in case of an urgent fix you are not controlling the delivery of the fix to your app (external dependency)	- fully isolated, so each component should handle all the possible modifications - need more resources and time to develop components

Table 1: Compare headless components and web-components

Usually, an enterprise solution is based on web components, as this solution is more secure and flexible, and can utilize all the possible requirements. At the same time, it is acceptable to use an existing library in enterprise solutions, it all depends on your needs and what risks you can accept.

The following headless components can be used:
- Material-UI base components: https://base-ui.com/react/overview/quick-start
- headlessui: https://headlessui.com/ - I will use this next in examples

2.2.2 Styling of the components

There are two popular methods of making stylings:
- Use tailwindcss
  - plus: the ready solution that already supports variables and will help to adopt easy
  - minus: styling web applications with such styling can be very complicated and look massive
- Use classic styling (css/less/sass etc.)
  - plus: no need for additional learning as it default skill
  - minus: need to develop on your own, but there is not much thing to do

Both solutions are fine and used across various companies and teams, I prefer to use styles as it is common.

2.2.3 First component

Using headless ui library and SASS make a common component that is styled as you need. Component code with typescript[4]:

```
import { Button as HeadlessButton } from "@headlessui/react";
import "./Button.scss";

export interface ButtonProps {
  label: string;
  onClick?: () => void;
  disabled?: boolean;
  variant?: "primary" | "secondary" | "danger";
}

const Button: React.FC<ButtonProps> = ({
  label,
  onClick,
  disabled,
  variant = "primary",
}) => {
  return (
        <HeadlessButton
        onClick={onClick}
        disabled={disabled}
        className={`cc-button m-${variant} ${disabled ? "m-disabled" : ""}`}
        >
        {label}
        </HeadlessButton>
  );
};

export default Button;
```

2.3 Centralised Showcase

It is common practice to use storybook, it already handles the structure for components, and supports any components properties changes[5].

Storybook should include Components and their usage.

Example of button showcase story with storybook[4]:

```
import React from "react";
import { Meta, StoryFn } from "@storybook/react";
import { Button, ButtonProps } from "../components";

export default {
  title: "Components/Button",
  component: Button,
  argTypes: {
        // Set up control types for props you'd like to modify in Storybook
        variant: {
        control: {
        type: "select",
        options: ["primary", "secondary", "danger"],
        },
        },
        disabled: {
        control: "boolean",
        },
        label: {
        control: "text",
        },
        onClick: {
        action: "changed",
        table: { disable: true },
        },
  },
} as Meta<typeof Button>;

// Template for rendering your Button
const Template: StoryFn<ButtonProps> = (args) => <Button {...args} />;

export const Primary = Template.bind({});
Primary.args = {
  label: "Primary Button",
  variant: "primary",
  disabled: false,
};

export const Secondary = Template.bind({});
Secondary.args = {
  label: "Secondary Button",
  variant: "secondary",
  disabled: false,
};

export const Danger = Template.bind({});
Danger.args = {
  label: "Danger Button",
  variant: "danger",
  disabled: false,
};

export const Disabled = Template.bind({});
Disabled.args = {
  label: "Disabled Button",
  variant: "primary",
  disabled: true,
};
```

3. Using Figma Design Tokens

Tokens are used to keep UI consistency between different pages. From page to page, there are the same padding and margins between elements and content, between header and content, between inputs, etc. Figma tokens don't support all the features that are needed for a project, see the comparison table 2 [6]:

Feature	**Figma Tokens (Native)**	**Tokens Studio Plugin**
Color Tokens	☐ Yes	☐ Yes
Typography Tokens	☐ No	☐ Yes
Spacing Tokens	☐ No	☐ Yes
Shadow Tokens	☐ No	☐ Yes
Border Tokens	☐ No	☐ Yes
Opacity Tokens	☐ No	☐ Yes
Radius Tokens	☐ No	☐ Yes
Sizing Tokens	☐ No	☐ Yes
Composite Tokens	☐ No	☐ Yes (e.g., Semantic Tokens)
Token Relationships	☐ No	☐ Yes
Dynamic Values	☐ No	☐ Yes
Versioning	☐ No (file-level)	☐ Yes
External Syncing	☐ No	☐ Yes (e.g., GitHub)
Custom Export Formats	☐ No	☐ Yes
Conditional Tokens	☐ No	☐ Yes
Integrated Experience	☐ Yes	☐ No (plugin required)
Learning Curve	☐ Low	☐☐ Moderate to High
Native Support in Figma designs	☐ Yes	☐ No

Table 2: Compare Figma tokens and Tokens Studio tokens

Tokens should be imported to the project from Figma on demand. The UX team exports tokens to the git repository and on each change, new tokens should pass quality gates.

3.1 Storing tokens

When we are talking about token storing, we should define the source of the tokens. The only source of truth for tokens is Figma and every time when we get new tokens — we generate them from what Figma tokens provides (or Token Studio, depending on what is chosen).

Tokens should be exported to a repository of web applications or to separate repositories that expose npm packages. By default, you can export it in code JS/TS, to use in styles CSS/SCSS/LESS/etc. This will allow you to use tokens across the application in different scenarios.

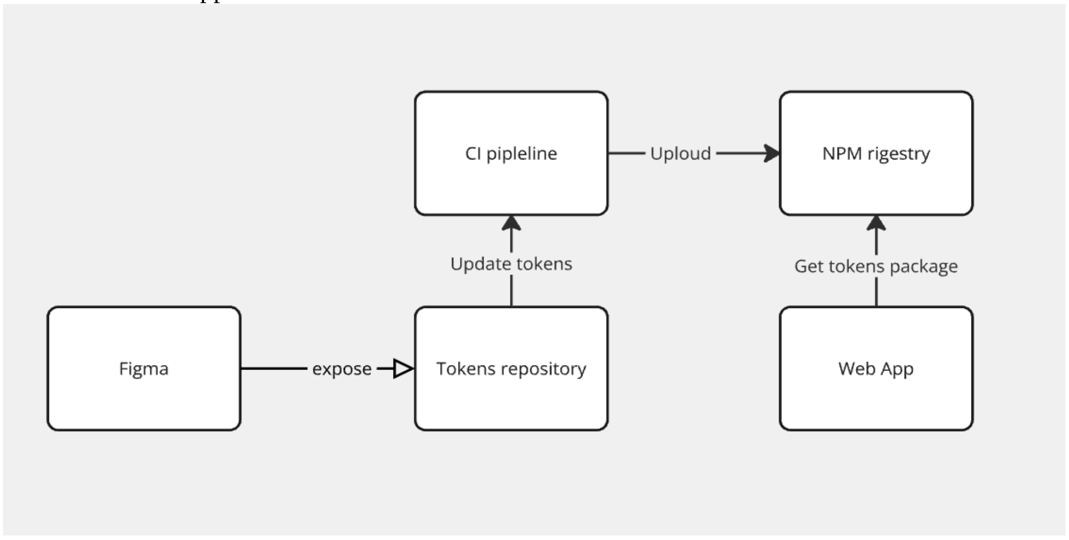

Fig. 1: Tokens export flow

3.2 Exported tokens

When decided by place of use, there are 3 types of tokens:

- Global – common tokens that can be used anywhere;
- Components tokens – those tokens are mostly based on common tokens but also can be unique. This is important as based on components tokens the app is defined.
- Local tokens – optional tokens that can be made if you want to fully control the UI from Figma, need to understand that if there are no plans to experiment with UX and changes, there is no need to make this token. By default, creating these tokens is not recommended.

Tokens by type:

- Color – tokens that contain colors;
- Spaces – margin and paddings between elements
- Sizing – font size, width and height of different elements
- Typography – typography tokens
- Theme – tokens with value that are dependent on the theme value

Example of tokens[4]:

```
/* src/tokens/global.css */
:root {
  /* COLORS */
  --color-white: #fff;
  --color-gray-100: #f9f9f9;
  --color-gray-300: #d1d5db;
  --color-gray-400: #ccc;
  --color-gray-500: #ddd;
  --color-gray-600: #eee;
  --color-grey-999: #999;
  --color-gray-hover: #999;
  --color-green-500: #4caf50;
  --color-blue-500: #3b82f6;
  --color-blue-50:  #eff6ff;
  --color-blue-100: #e6f7ff;
  --color-blue-600: #2563eb;
  --color-blue-700: #1d4ed8;
  --color-blue-800: #1e40af;
  --color-sky-100:   #e0f2fe;
  --color-sky-200:   #bae6fd;
  --color-red-300: #dc2626;
  --color-blue-focus-shadow: rgba(59, 130, 246, 0.5);
  --color-green-focus-shadow: rgba(76, 175, 80, 0.5);
  --color-red-focus: #b91c1c;

  /* TYPOGRAPHY */
  --font-family-base: 'Arial, sans-serif';
  --font-size-base: 16px;
  --font-size-small: 14px;
  --font-size-large: 18px;
  --font-weight-normal: 400;
  --font-weight-bold: 600;

  /* SPACING */
  --spacing-xs: 4px;
  --spacing-sm: 8px;
  --spacing-md: 16px;
  --spacing-lg: 24px;
  --spacing-xl: 32px;

  /* THEME */
  --background-color__white: var(--color-white);
  --background-color__dark: var(--color-white);
}
```

3.3 Theme tokens

To support the theme tokens you need to define the behavior in the application. Tokens set the values and the app declares the tokens and how/ehn to change the value of the token. The simplest way to do it is to inject tokens into the root of the app. See containerStyle object that is added to the DOM[4].

```
function App() {
  const [toDoItems, setToDoItems] = useState<ToDoItemType[]>(initialToDoItems);
  const [theme, setTheme] = useState<"white" | "dark">("white");

  const toggleItem = (id: number) => {
        setToDoItems((prevItems) =>
        prevItems.map((item) =>
        item.id === id ? { ...item, completed: !item.completed } : item
        )
        );
  };

  const containerStyle: object = {
        "--background-color":
        theme === "white" ? "var(--color-white)" : "var(--color-grey-999)",
  };

  return (
        <section style={containerStyle} className="app-body">
        <Layout padding="large" border={true}>
        <h1>My ToDo List</h1>
        <Button
```

```
            label={theme}
            onClick={() =>
            setTheme((theme) => (theme === "white" ? "dark" : "white"))
            }
            />
            <ToDoList items={toDoItems} onToggle={toggleItem} />
            </Layout>
            </section>
  );
}

export default App;
```

4. Automation and Unit Testing

4.1 Automated tests

When writing automated tests, you need to keep in mind the running time of the tests and the number of tests. If you test running for a long time — it affects your delivery. When making a decision, you need to consider other existing steps in the CI/CD process and find your balance. Up to 15 minutes should be enough to take all types of tests. Tests should run in parallel on all existing cores until there is enough RAM.

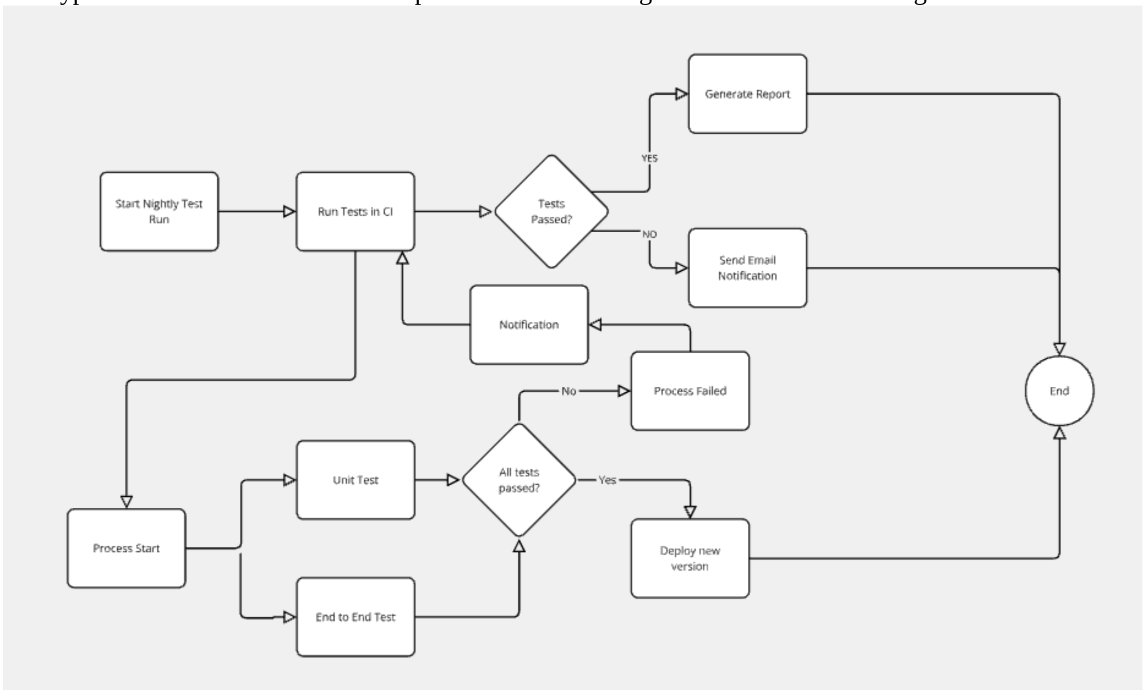

Fig. 2: Tests run flow

The main purpose of making automated tests is to minimize the amount of bugs by testing in real browsers. Tests amount should be minimised and tests should cover the functionality by type.

Test types that can be run in the browser:

4.1.1 A11y

Those types of tests should show the problems with accessibility with UI.

```
import { test } from "@playwright/test";
import { injectAxe, checkA11y } from "axe-playwright";

test.describe("SearchInput Accessibility Tests", () => {
  test.beforeEach(async ({ page }) => {
            await page.goto("/search-input/default");
            await injectAxe(page);
  });
  test("should have no accessibility violations", async ({ page }) => {
            await checkA11y(page);
  });
});
```

4.1.2 Screenshot testing

Those types of tests should not contain much logic, those tests need to fix the component UI state to avoid any disputes regarding margins/font sizes/colors, etc.

```
import { test, expect } from "@playwright/test";

test.describe("SearchInput visual tests", () => {
            test('should render with default placeholder', async ({ page }) => {
            await page.goto('/search-input/default');
            expect(await page.screenshot()).toMatchSnapshot();
            });

            test("should render combobox with variants", async ({ page }) => {
            await page.goto('/search-input/with-items');
            const input = await page.getByPlaceholder("start typing...");
            input.click();
            input.fill('first');
            expect(await page.screenshot()).toMatchSnapshot();
            });
})
```

- Feature testing

```
import { test, expect, Page } from "@playwright/test";

async function getSearchInputOptionsCount(page: Page) {
            return await page.locator('.cc-searchinput-option').count()
}

test.describe('SearchInput feature tests', () => {
            test('should filter combobox items depending on input', async ({ page }) => {
            await page.goto('/search-input/with-items');
            const input = await page.getByPlaceholder("start typing...");
            await input.click();

            await input.fill('first');
            expect(await getSearchInputOptionsCount(page)).toBe(1);

            await input.click();
            await input.fill('');
            expect(await getSearchInputOptionsCount(page)).toBe(3);

            await page.locator('.cc-searchinput-option').nth(1).click();
            expect(input).toHaveValue('two');
            });
})
```

4.1.3 Touchscreen tests

Those tests are focused on touch events, when you need to test the screen toiuching and be sure that it will work on touch devices. Can be added the viewport parameter to make the screen smaller for full emulation of mobile.

```
import { test, expect } from "@playwright/test";
```

```
test.describe('SearchInput touch', () => {
	test('should input data', async ({page}) => {
	await page.goto('/search-input/default');
	const input = await page.getByPlaceholder("start typing...");
	const inputBox =  await input.boundingBox();

	if (!inputBox) {
	throw new Error("Can not find SearchInput");
	}

	await page.touchscreen.tap(
	inputBox.x + inputBox.width / 2,
	inputBox.y + inputBox.height / 2
	);

	await page.keyboard.type('test');

	expect(await page.getByText("test")).toBeDefined();
	})
})
```

4.2 Unit tests

Make simple unit tests and only then integration unit tests. By the integration unit tests, I mean tests that cover the logic of the current component and its dependencies. When we are talking about testing components then there is no need to test common components, you can

1) Use dependencies as a block box

Do not use internal implementation of your function or common component to test. Ideally mock components and **public** methods, mocking private methods breaking encapsulation.

2) Don't test dependencies

You already should have tests to test your component, while testing your root component no need to test all the common components features.

Example of tests[4]

```
// ToDoList.test.tsx
import React from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { ToDoList, ToDoListProps } from "./ToDoList";

describe("ToDoList", () => {
  const itemsMock = [
	{ id: 1, text: "Buy groceries", completed: false },
	{ id: 2, text: "Walk the dog", completed: false },
	{ id: 3, text: "Read a book", completed: false },
  ];

  const renderToDoList = (props: Partial<ToDoListProps> = {}) => {
	const defaultProps: ToDoListProps = {
	items: itemsMock,
	onToggle: jest.fn(),
	...props,
	};
	return render(<ToDoList {...defaultProps} />);
  };

  it("displays all todo items by default", () => {
        renderToDoList();

        expect(screen.getByText("Buy groceries")).toBeDefined();
        expect(screen.getByText("Walk the dog")).toBeDefined();
        expect(screen.getByText("Read a book")).toBeDefined();
  });

  it("filters items when selecting a search suggestion", async () => {
        renderToDoList();
        const user = userEvent.setup();

        // Type enough to show "Walk the dog" as a suggestion
        const searchInput = screen.getByPlaceholderText("Search for to do item...");
        await user.type(searchInput, "Walk");

        // Grab the combobox option by role and name
        const suggestion = await screen.findByRole("option", {
        name: /walk the dog/i,
        });
        await user.click(suggestion);

        // Now only that item is shown
        expect(screen.getByText("Walk the dog")).toBeDefined();
        expect(screen.queryByText("Buy groceries")).not.toBeDefined();
        expect(screen.queryByText("Read a book")).not.toBeDefined();
  });

  it("calls onToggle when an item is toggled", async () => {
        const onToggleMock = jest.fn();
        renderToDoList({ onToggle: onToggleMock });
        const user = userEvent.setup();

        const itemToToggle = screen.getByText("Buy groceries");
        await user.click(itemToToggle);

        expect(onToggleMock).toHaveBeenCalledTimes(1);
        expect(onToggleMock).toHaveBeenCalledWith(1);
  });
});
```

5. Process additions

The Following practices should be integrated into the teamwork process:

1) The UX team should keep the dev team informed about every change in common components. Every change in Figma should be discussed with developers.
2) The UX team should keep components with the same properties as in development. Each common component feature should be described in Figma.
3) The dev team should notify about each change in components when they are made by dev team initiatives.
4) The dev team should make a demo for the UX team on each change for quality control.

6. Benefits and Challenges

6.1 Benefits

- **Re-usable common components that fast adapt to UX changes:**

According to SDLC[7], before the development phase goes to the design stage. By keeping synchronised implementation with designs, you ensure that the development team always makes the same UX prototype, thereby minimizing development time. At the same time, the UX team can test assumptions without a development team.

- **Manage the work with different projects and help to make them branded:**

When multiple projects belong to the same brand, it is expected that they have the same UX design code. No need to make extra common components for each project, it is easier to make it reusable for several projects and at the same time keep similar experiences from project to project.

- **Independent functional components:**

As components make it easy to adopt new changes, it is better to add functionality for components and to support them on a common level. There is no need to add styles that rewrite components.

6.2 Challenges

- **The test should be well-planned to avoid endless CI process:**

The execution time of tests should not be too long as we want to run tests as often as possible. Double testing should be avoided, splitting tests in types should help to make tests smaller and more focused on features functionality.

- **Continues communication is key for making all work:**

The UX team should synchronize and have regular update meetings with the development team. Sometimes changes come from the development team and changes should be directed to the UX team, as UX designs are a single source of truth.

- **A skilled and involved dev team is needed for making components:**

While web pages can be made by less experienced developers or even by back-end developers, components should be implemented by skilled front-end developers that are skilled and deep dive into the context of components problems. Usually, the team that is working on components is called a “platform team”. The platform team should decide and define what is UI logic, what should be in components and what parts of business logic should remain separate. Reusable scenarios can be placed in separate shared reusable functions, but not in the components, as components are dummies.

- **Organise contributions to components from other teams:**

The amount of changes can be big, when there is a big amount of people working with components. Need to organize the process of accepting changes to components with automated tests on CI.

7. Conclusion

The article provides a comprehensive approach to building scalable design systems for enterprise applications by integrating design tools like Figma with development practices such as reusable UI components and web components with React. The system can adopt a large number of changes and keep the desired quality of the changes in large projects that are usually used in enterprises.

8. References

[1]. *Understanding and supporting the design systems practice, Empirical Software Engineering*, 27, 2022, article 146. [Online]. Available: https://link.springer.com/article/10.1007/s10664-022-10181-y?utm_source=chatgpt.com

[2]. *Material UI components,* Google documentation. [Online]. Available: https://m3.material.io/components/snackbar/overview

[3]. *Resources for Developers, by Developers; Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_components

[4]. A. Marozau, *Showcase git repository*, 2025 [Online]. Available: https://github.com/MarozauArtsiom/showcase-app

[5]. *Storybook: Official Documentation,* 2025. [Online]. Available: https://storybook.js.org/docs/react

[6]. A. Marozau, *Simplifying UX Updates in Legacy Applications with Tokens: Figma vs. Tokens Studio*, Hackernoon, 16 October 2024. [Online]. Available: https://hackernoon.com/simplifying-ux-updates-in-legacy-applications-with-tokens-figma-vs-tokens-studio

[7]. G. Jackson, *What is the software development lifecycle (SDLC)?* IBM, 13 December 2024. [Online]. Available: https://www.ibm.com/think/topics/sdlc