


Iterate your code with AI

 developers.figma.com/docs/code/iterate-your-code-with-ai

- 
- Code
- Guides
- Iterate your code with AI

You can bring your own React code into Figma Make or Figma Sites and iterate on it using AI in the code editors.

Figma Make

In Figma Make, there are a few ways to bring your own code. In the code editor, you can:

- Paste your React code into `App.tsx` (the default file for Figma Make web apps and functional prototypes)
- If your React app consists of multiple files, instruct the model to create empty or mock versions of the files, and then paste your React code into those files
- Paste your code directly into the AI chat and instruct the model to use it

Once your code is functional in Figma Make, you can then improve it, such as using the chat to add functionality or change the styling.

Example: Task manager app

The following example is a very simple React app for tracking tasks. It consists of three files: `App.tsx`, `TaskForm.tsx`, and `TaskList.tsx`. Here's the example code, followed by prompts we'd use to work with the code in Figma Make.

App.tsx

`App.tsx` describes our task manager app, including some starting default tasks, and implements the form and list components we define in the other files.

```

import React, { useState } from "react";
import TaskForm from "../TaskForm";
import TaskList from "../TaskList";

export default function App() {
  const [tasks, setTasks] = useState([
    { id: 1, text: "Prepare quarterly report", completed: true },
    { id: 2, text: "Schedule team meeting", completed: false }
  ]);

  function addTask(text) {
    const newTask = {
      id: Date.now(),
      text: text,
      completed: false
    };
    setTasks([...tasks, newTask]);
  }

  function toggleTask(id) {
    setTasks(
      tasks.map(task =>
        task.id === id ? { ...task, completed: !task.completed } : task
      )
    );
  }

  return (
    <div className="task-app">
      <h1>Task Manager</h1>
      <TaskForm onAddTask={addTask} />
      <TaskList tasks={tasks} onToggleTask={toggleTask} />
    </div>
  );
}

```

TaskForm.tsx

TaskForm.tsx creates the form element that's used to add new tasks to the list.

```

import React, { useState } from "react";

export default function TaskForm({ onAddTask }) {
  const [text, setText] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    if (text.trim()) {
      onAddTask(text);
      setText("");
    }
  }

  return (
    <form onSubmit={handleSubmit} className="task-form">
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add a new task"
      />
      <button type="submit">Add</button>
    </form>
  );
}

```

TaskList.tsx

TaskList.tsx is the list of tasks in our task manager app.

```

import React from "react";

export default function TaskList({ tasks, onToggleTask }) {
  if (tasks.length === 0) {
    return <p>No tasks yet. Add one above!</p>;
  }

  return (
    <ul className="task-list">
      {tasks.map(task => (
        <li
          key={task.id}
          className={task.completed ? "completed" : ""}
          onClick={() => onToggleTask(task.id)}
        >
          <span>{task.text}</span>
          <span className="status">
            {task.completed ? "✓" : "○"}
          </span>
        </li>
      ))}
    </ul>
  );
}

```

For all the following approaches, let's assume we've already created a Figma Make file that will contain our app. By default, in the code editor, `App.tsx` is selected and contains template code.

Approach 1: Paste the code into the `App.tsx` file in Figma Make

To get our app working in Figma Make, we paste the content of each file into `App.tsx` in the code editor. This results in some errors because we have multiple `import React from "react";` statements, and because we're trying to import from `TaskList.tsx` and `TaskForm.tsx`, which don't exist in the context of our Figma Make file.

After pasting the code, we use the following prompt in the chat: `Fix the errors in the code. Do not change the functionality or styling of the app.`

Figma Make reviews the code, removes the extraneous import statements, and fixes the component references, which renders the app functional.

Approach 2: Prompt Figma Make to create the corresponding files

We know that we have three files, `App.tsx`, `TaskList.tsx`, and `TaskForm.tsx` that we rely on to build our app. `App.tsx` already exists with some template code in Figma Make, so we don't need to worry about creating that one.

To get started, we use the following prompt in the chat: `Create two empty files on the same level as App.tsx: TaskList.tsx and TaskForm.tsx`

The model generates two empty files for us that exist in the same directory as `App.tsx`. Then, we paste the code from each of our files into the corresponding files. That's it! The app is working in Figma Make.

If we did encounter any errors, we'd use the prompt in Approach 1 to resolve them.

Approach 3: Paste the code directly into the chat

To get started, we decide we want to skip the code editor completely. Instead, in the chat, we use the following prompt:

I want to use my own code for this app. It's in three files:

App.tsx:

<paste code from App.tsx>

TaskList.tsx:

<paste code from TaskList.tsx>

TaskForm.tsx:

<paste code from TaskForm.tsx>

Do not make functional or style changes to the app.
I want to use my code as-is, but resolve any errors.

Figma Make reviews and implements the code. Because it's a non-deterministic system, one of two things will likely happen:

- Figma Make will create two new files, `TaskList.tsx` and `TaskForm.tsx`. The files may be at the same level as `App.tsx`, or added to a new or existing directory, such as `./components`.
- Figma Make will combine the code and put it all into `App.tsx`.

Next: Improve the app

Our task manager app is very basic. It lacks styling, and some common functionality.

First, we decide we want to be able to edit and delete tasks. We use the following prompt:
`Add the ability to edit and delete tasks to my app`

Maybe we also want the ability to have multiple task lists and store them for awhile. We use the following prompt: `Make it so I can create new task lists, as well as view and delete existing task lists. Use browser storage so the task lists aren't lost when I leave the page.`

Finally, we want to style our app so it's a little more friendly and usable. We use the following prompt: `Style my app. Give it a nice, clean, modern look.`

There we go! We have a styled, more functional app based on the code we original created outside of Figma Make.

Figma Sites

In Figma Sites, you can bring your React code into code components or layers. Code components and layers in Figma Sites are best used to capture individual parts of UI rather than a whole web app. For example, code components are great for standalone UI

elements like animated text, counters, stylized clocks, and interactive charts, among other possibilities.

First, create a code component or code layer, then open the code editor to make changes to it.

In the code editor, you can:

- Paste your React code directly
- If your React component relies on multiple files, paste the code for all the necessary files into the editor and then ask the model to fix the errors
- Paste your code directly into the AI chat and instruct the model to use it for the component or layer

After your code is in the editor and functional, you can use the chat to continue to improve the UI element.

Example: Lo-fi text shader

The following example is a React component that creates a lo-fi animated shader effect for a string of text. Here's the example code, followed by some ways we'd approach using it and improving it in Figma Sites.

```

import React, { useRef, useEffect } from "react";

export default function TextShader() {
  const canvasRef = useRef<HTMLCanvasElement | null>(null);
  const containerRef = useRef<HTMLDivElement | null>(null);
  const TEXT = "Example", FONT = 64, SPEED = 0.5, SHIFT = 0.6;

  useEffect(() => {
    const canvas = canvasRef.current, container = containerRef.current;
    if (!canvas || !container) return;
    const ctx = canvas.getContext('2d');
    if (!ctx) return;

    const resize = () => {
      if (container && canvas) {
        canvas.width = container.clientWidth;
        canvas.height = container.clientHeight;
      }
    };
    resize();
    window.addEventListener('resize', resize);

    const noise = (x, y) => Math.sin(x * 0.1) * Math.cos(y * 0.1) * 0.5 +
      Math.sin(x * 0.4 + y * 0.3) * Math.cos(y * 0.2 - x *
0.1) * 0.5;

    let animationId: number;
    const primaryColor = [0, 198, 255], secondaryColor = [0, 114, 255];
    const pixelSize = 2, intensity = 20;

    const animate = (timestamp: number) => {
      const time = timestamp * 0.001 * SPEED;
      ctx.clearRect(0, 0, canvas.width, canvas.height);

      const centerX = canvas.width / 2, centerY = canvas.height / 2;

      const offscreen = document.createElement('canvas');
      offscreen.width = canvas.width;
      offscreen.height = canvas.height;
      const offCtx = offscreen.getContext('2d');
      if (!offCtx) return;

      offCtx.font = `bold ${FONT}px sans-serif`;
      offCtx.textAlign = 'center';
      offCtx.textBaseline = 'middle';
      offCtx.fillStyle = "#ffffff";
      offCtx.fillText(TEXT, centerX, centerY);

      const metrics = offCtx.measureText(TEXT);
      const textWidth = Math.ceil(metrics.width);
      const textHeight = Math.ceil(FONT * 1.2);

      const x1 = Math.max(0, centerX - textWidth/2 - 10);
      const y1 = Math.max(0, centerY - textHeight/2 - 10);
      const width = Math.min(canvas.width - x1, textWidth + 20);
      const height = Math.min(canvas.height - y1, textHeight + 20);

```

```

const imageData = offCtx.getImageData(x1, y1, width, height);

for (let y = 0; y < height; y += pixelSize) {
  for (let x = 0; x < width; x += pixelSize) {
    const i = (y * width + x) * 4 + 3;

    if (textData.data[i] > 0) {
      const worldX = x1 + x, worldY = y1 + y;

      const distortX = noise(worldX * 0.05, worldY * 0.05 + time) *
intensity;
      const distortY = noise(worldX * 0.05 + 4000, worldY * 0.05 + time *
1.5) * intensity;

      const colorFactor = (noise(worldX * 0.01 + time, worldY * 0.01) * 0.5
+ 0.5) * SHIFT + (1 - SHIFT) * 0.5;
      const r = Math.floor(primaryColor[0] + (secondaryColor[0] -
primaryColor[0]) * colorFactor);
      const g = Math.floor(primaryColor[1] + (secondaryColor[1] -
primaryColor[1]) * colorFactor);
      const b = Math.floor(primaryColor[2] + (secondaryColor[2] -
primaryColor[2]) * colorFactor);

      ctx.fillStyle = `rgb(${r}, ${g}, ${b})`;
      ctx.beginPath();
      ctx.arc(worldX + distortX, worldY + distortY, pixelSize, 0, Math.PI *
2);

      ctx.fill();
    }
  }
}
animationId = requestAnimationFrame(animate);
};

animationId = requestAnimationFrame(animate);
return () => {
  cancelAnimationFrame(animationId);
  window.removeEventListener('resize', resize);
};
}, []);

return (
  <div ref={containerRef} className="w-full h-full flex items-center justify-
center">
    <canvas ref={canvasRef} className="w-full h-full" />
  </div>
);
}

```

For all the following approaches, let's assume we've created a code component or added a code layer to the canvas. By default, the code editor contains some template code for our code component or layer.

Approach 1: Paste the code into the code editor

To get started using our text shader, we paste the code into the code editor. There might not be any errors, but just in case, we'll use the following the prompt in the chat: **Fix any errors in my code. Do not change the functionality or styling of the component, I want to use it as-is. If there aren't any errors, just respond 'Looks good' and do not make any changes.**

We're all set to improve our code.

Approach 2: Paste the code directly into chat

To get our animated text going without using the editor, we paste our whole component directly into chat using the following prompt:

Here's my component:

<paste text shader code>

If there are any errors fix them,
but don't make any other changes to the code or styling.
Besides fixing the errors, I want to use the code as-is.

Figma Sites moves our code into the code component or layer, fixing any errors it encounters. We're ready to improve the code.

For both approaches, if we did this in a code component and want to preview it, we'll click **Add to canvas** in the upper-right corner of the editor, and then click where we want the text to appear. If it's in a code layer, we're already all set.

When we preview the site, we're able to see the animated text.

Next: Improve the code

It's great that our animated text is working, but now we want to make some improvements.

First, we like the effect, but everything is hardcoded. It's not easy to change the string of text or parameters of the animation, such as how fast it moves or the colors. We use the following prompt: **Add props to my component for text, colors, and other parts of the animation.** Now when we reuse the component or layer in different places on our site, we can make it say different things or change the way it looks.

Maybe we want to play with the animation some more. Lo-fi is great, but we want try some other styles. We use the following prompt: **Improve the shader effect so it feels fancier and more liquid**

We can continue to play with and iterate on our code in this way until we find the style that best fits what we're going for.