Code execution breakdown

Excellent question, Vipul 👏 — you've spotted a **very deep and important concept** in JavaScript evaluation order.

Let's carefully unpack what's happening here step-by-step — because the key point is **when `console.log(i)`** runs.

🧠 Your code:

```
js

const ids = [];
for (let i = 0; i < 3; i++) {
  ids.push(console.log(i));
}
```

🔍 Step-by-step: What JavaScript actually does

When JavaScript executes a line like:

```
js

ids.push(console.log(i));
```

it doesn't call **`push()`** first.
It must **evaluate the argument inside the parentheses** *before* calling the function.

So the order is:

1. **Evaluate `console.log(i)`**
    - That means: call **`console.log()`** immediately.
    - This prints **`i`** to the console right away.
    - Then it returns **`undefined`**.
2. **Take the return value (`undefined`)**
    - and pass that to **`ids.push(...)`**.
3. **Call `ids.push(undefined)`**
    - That adds **`undefined`** to the array.

🧩 Visual timeline

Step	Expression	Action	Result
1	**`console.log(i)`**	Prints **`i`** immediately	returns **`undefined`**
2	**`ids.push(undefined)`**	Adds **`undefined`** to array	—

🧠 Key Concept: Function argument evaluation order

In JavaScript (and most programming languages):

> Before a function is called, all its arguments are **evaluated first**.

So in:

```js
ids.push(console.log(i));
```

JS first runs **`console.log(i)`** (which prints something)
then runs **`.push()`** with whatever that returned (which is **`undefined`**).

✅ Hence the output:

```javascript
0
1
2
[undefined, undefined, undefined]
```

You're not "calling from ids" — you're calling **`console.log()`** *before* the push happens because JS must know what value to pass into **`push()`**.

Would you like me to show a **visual analogy** (like a timeline diagram) of how the JS engine executes that line internally? It'll make this even clearer.