Excellent topic choice 👍

Now I'll **rewrite Microtasks from scratch**, keeping your content **but upgrading it** to:

- ✅ **engine-level accuracy**
- ✅ **clear mental models**
- ✅ **event-loop timeline clarity**
- ✅ **interview-ready traps**
- ✅ **simpler + deeper at the same time**

This is **one of the most important JS internals topics**. Many devs *think* they know it — few actually do.

🔬 Microtasks in JavaScript — INTERNALS & EVENT LOOP DEEP DIVE

1️⃣ First: One Critical Correction (Very Important)

> ❌ "Microtasks run after the function or program completes"

✅ Correct Statement

Microtasks run:

- After the **current call stack becomes empty**
- **Before any macrotask**
- **Immediately**, in the same event loop turn

Microtasks are **NOT delayed tasks** — they are **high-priority continuation jobs**.

2️⃣ JavaScript Execution Model (Foundation)

JavaScript has **only one thread**, but it coordinates work using:

```mathematica
┌─────────────┐
│ Call Stack  │ ← executes sync code
└─────────────┘
       ↓
┌─────────────┐
│ Microtask   │ ← promises, queueMicrotask
│ Queue       │
└─────────────┘
       ↓
┌─────────────┐
│ Macrotask   │ ← setTimeout, events, I/O
│ Queue      │
└────────────┘
```

⚠️ **Order is strict and guaranteed**

3️⃣ What Exactly Is a Microtask?

✅ Definition (Precise)

A **microtask** is a **job scheduled by the JS engine** to:

- Continue promise execution
- Resume async functions
- Run *before* browser rendering and macrotasks

Common Sources of Microtasks

Source	Why
`Promise.then()`	promise continuation
`Promise.catch()`	error continuation
`Promise.finally()`	cleanup
`await`	async function resume
`queueMicrotask()`	explicit microtask

Interview Trap ❗

❌ `setTimeout` creates microtask
✅ `setTimeout` creates **macrotask**

4️⃣ What Is a Macrotask?

Macrotasks represent **new, separate events**:

Source	Type
`setTimeout`	Timer
`setInterval`	Timer
DOM events	UI events
Network I/O	Fetch callbacks

Source	Type
`setImmediate` (Node)	Check phase

They run **after microtasks are fully drained**.

5️⃣ Event Loop — REAL Algorithm (Simplified)

This is what the JS engine **actually does**:

```text
while (true) {
  execute all sync code (call stack)

  while (microtask queue not empty) {
    run microtask
  }

  run ONE macrotask

  render UI (browser only)
}
```

⚠️ Microtasks are drained completely before macrotasks

6️⃣ Microtask Example (Promise)

```js
console.log("Start");

Promise.resolve().then(() => {
  console.log("Microtask");
});

console.log("End");
```

Output

```powershell
Start
End
```

```
Microtask
```

Why This Happens (Internals)

1. `Start` → call stack
2. `Promise.then` → microtask queue
3. `End` → call stack
4. Call stack empty
5. Microtask queue executed

🔑 Key Rule

> **Promise callbacks never interrupt sync code**

7️⃣ Macrotask Example (`setTimeout`)

```js

console.log("Start");

setTimeout(() => {
  console.log("Macrotask");
}, 0);

console.log("End");
```

Output

```powershell

Start
End
Macrotask
```

Why Delay = 0 Still Delays?

Because:

- Timer callback is **queued**
- Must wait for:
  - call stack
  - microtasks
  - event loop turn

8️⃣ Microtask vs Macrotask (Together)

js

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise");
});

console.log("End");
```

Output

powershell

```
Start
End
Promise
Timeout
```

Internal Timeline

Step	Queue
Start	Call stack
Timeout	Macrotask queue
Promise	Microtask queue
End	Call stack
Promise	Microtask executes
Timeout	Macrotask executes

9️⃣ `await` = Microtask (Very Important)

js

```
async function test() {
  console.log("A");
  await null;
  console.log("B");
}

test();
console.log("C");
```

Output

```css
A
C
B
```

Why?

- `await` splits function
- Code after await → microtask

Equivalent internal rewrite:

```js
Promise.resolve(null).then(() => {
  console.log("B");
});
```

🔟 `queueMicrotask()` — Explicit Control

```js
console.log("Start");

queueMicrotask(() => {
  console.log("Microtask");
});

setTimeout(() => {
  console.log("Macrotask");
}, 0);

console.log("End");
```

Output

```powershell
Start
End
Microtask
Macrotask
```

Why `queueMicrotask` Exists

- Safer than `Promise.resolve().then`
- No promise allocation
- Used internally by browsers

1️⃣1️⃣ Infinite Microtask Starvation (Danger ⚠️)

```js
function loop() {
  queueMicrotask(loop);
}
loop();
```

What Happens?

- Microtask queue never empties
- Macrotasks never run
- UI freezes

Interview Trap ❗

Q: Can microtasks block rendering?

A: ✅ Yes (starvation)

1️⃣2️⃣ Browser Rendering Order

In browsers:

```nginx
JS execution
↓
Microtasks
↓
Render UI
```

```
↓
Macrotask
```

⚠️ Microtasks **run before repaint**

1️⃣3️⃣ Summary Table (Memorize)

Feature	Microtask	Macrotask
Priority	High	Low
Queue	Microtask queue	Task queue
Runs when	Stack empty	After microtasks
Examples	Promise, await	setTimeout
UI render	Before	After

🎯 INTERVIEW QUESTIONS (WITH ANSWERS)

Q1: Why does Promise run before setTimeout?

➡️ Microtask priority

Q2: Does await create a microtask?

➡️ Yes

Q3: Can microtasks block UI?

➡️ Yes (if infinite)

Q4: Does setTimeout(0) mean immediate?

➡️ No

Q5: Is microtask queue FIFO?

➡️ Yes

🧠 FINAL MENTAL MODEL (ONE LINE)

> **JavaScript finishes what it started, completes all promises, then handles timers.**

⏭️ NEXT TOPICS (Same Depth)

Choose ONE:

1️⃣ **Event Loop (browser vs Node.js differences)**

2️⃣ **Unhandled Promise Rejections**

3️⃣ **setTimeout vs setImmediate vs process.nextTick**

4️⃣ **Async rendering & UI blocking**

5️⃣ **Fetch + microtask scheduling**

Reply with the **number**.