

Replit end to end

Beginner to advance

Giving some Context

Pre requisites

1. Basic Coding (loops, if else, variables)
2. Node.js

Good to haves

1. Docker / Containerization
2. Kubernetes
3. AWS ASGs

What we'll learn

Basic

1. Backend communication
2. Docker / Containerization
3. Isolated environments
4. Remote code execution
5. [repl.it](#) system design/architecture

Advance

1. Kubernetes
2. Pseudo Terminals
3. Nix

Before we start - Disclaimer

We'll be taking 3 approaches to solve this problem

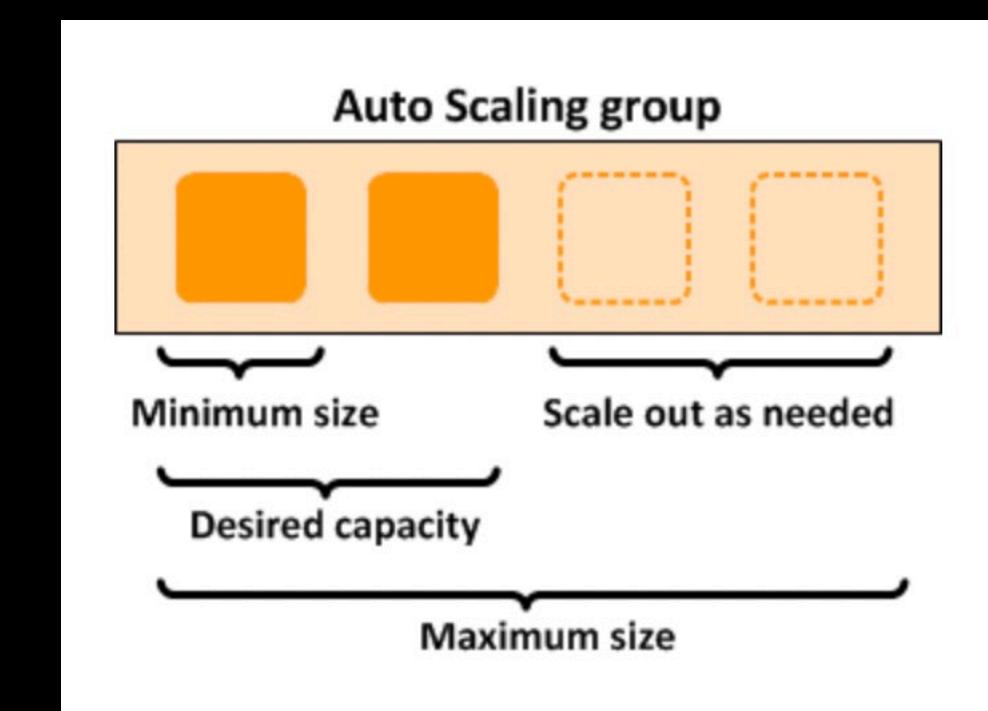
Beginner friendly

1. How I would implement it from first principles back in college
2. Highly **insecure**
3. If you use this approach in an interview you'll be rejected
4. Good to know what's happening though
5. **Doesn't scale**



Cloud specific Autoscaling constructs

1. Mid approach
2. Great for a new startup
3. Secure, autoscales
4. Uses cloud dependant constructs (ASG, ECS)

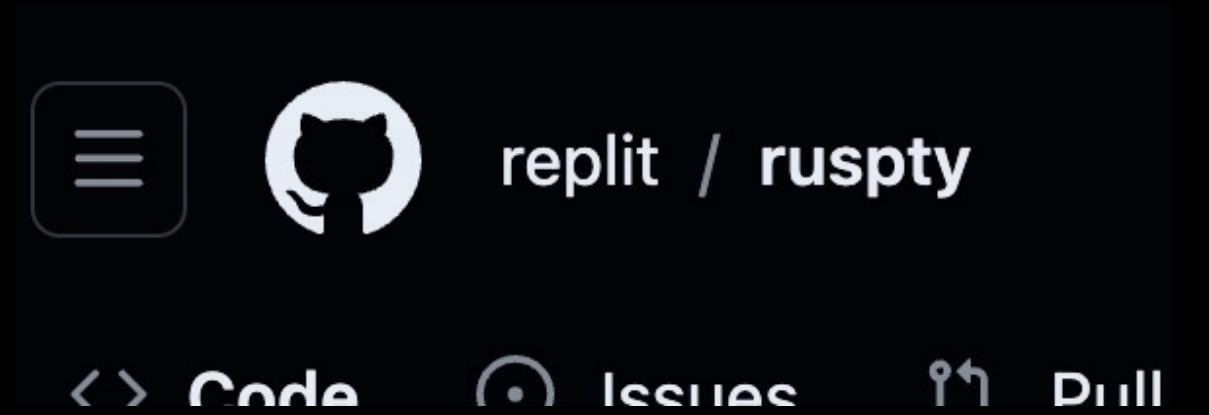


Cloud native Approach

1. Good approach
2. Autoscales, secure
3. How I would build replit



How did I think of this architecture?



First principles (~8 years of coding)
Reverse engineering repl.it (~3 hours)
Going through their blog posts, GitHub and videos on YT

All are linked in the description

- [Codecademy](#): Uses xterm.js in its courses on Bash.
- [Laravel Ssh Web Client](#): Laravel server inventory with ssh web client to connect at server using xterm.js
- [Replit](#): Collaborative browser based IDE with support for 50+ different languages.
- [TeleType](#): cli tool that allows you to share your terminal online conveniently. Show off mad cli-fu, help a col-
- [Intervue](#): Pair programming for interviews. Multiple programming languages are supported, with results dis-

[crosis](#) Public

A JavaScript client that speaks Replit's container protocol

TypeScript ⭐ 102 📂 19

What we're building

Online IDE for long running backend/frontend apps

The screenshot displays the Replit IDE interface, which integrates a code editor, a browser preview, and a terminal window.

Code Editor: The main area shows the `index.html` file content:<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="UTF-8" />
 <link rel="icon" type="image/svg+xml" href="/favicon.svg" />
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
 <title>React + TypeScript + Replit</title>
 </head>
 <body>
 <div id="root"></div>
 <script type="module" src="/src/index.tsx"></script>

 <!--
 This script places a badge on your repl's full-browser view
 back to your repl's cover
 page. Try various colors for the theme: dark, light, red,
 orange, yellow, lime, green,
 teal, blue, blurple, magenta, pink!
 -->
 <script src="https://replit.com/public/js/replit-badge-v2.js"
 theme="dark" position="bottom-right"></script>
 </body>
</html>

Browser Preview: The preview window shows the rendered output of the code, titled "React + Vite ⚡ + Replit". It includes a "Run on Replit" button.

Terminal: The terminal window shows the command history and output:> react-typescript@1.0.0 dev
> vite

VITE v3.0.4 ready in 2426 ms
→ Local: http://localhost:5173/
→ Network: http://172.31.196.31:5173/

File Explorer: The sidebar shows the project structure with files like `App.css`, `App.tsx`, `index.tsx`, and `index.html`.

What we're building

A screenshot of a developer environment interface. At the top, there's a toolbar with icons for Stop, Invite, Deploy, and Help. Below the toolbar, the main area has several tabs: 'index.ts' (active), 'Webview', 'Console', and 'Shell'. The 'index.ts' tab shows the following TypeScript code:

```
1 import express from "express";
2 const app = express();
3
4 app.get("/", (req, res) => {
5   res.send("hello world")
6 });
7
8 app.listen(3000)
```

The 'Webview' tab displays a simple 'hello world' response. The 'Console' tab shows npm output:

```
16 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New major version of npm available! 8.19.2 -> 10.4.
0
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v10.4.0>
npm notice Run `npm install -g npm@10.4.0` to update!
npm notice
```

The 'Shell' tab shows a command-line interface with a 'Run' button.

1. Online IDE for long running backend/frontend apps
2. Ability to start a fresh environment (in Rust, Go, Python, Node, React ...)
3. Ability to autoscale servers with users
4. Run code in an isolated environment

What we're NOT building, but good to haves

The screenshot shows the Replit login page. At the top left is the Replit logo. Below it, a dark green banner features the text "Make something great." and a bulleted list of five features:

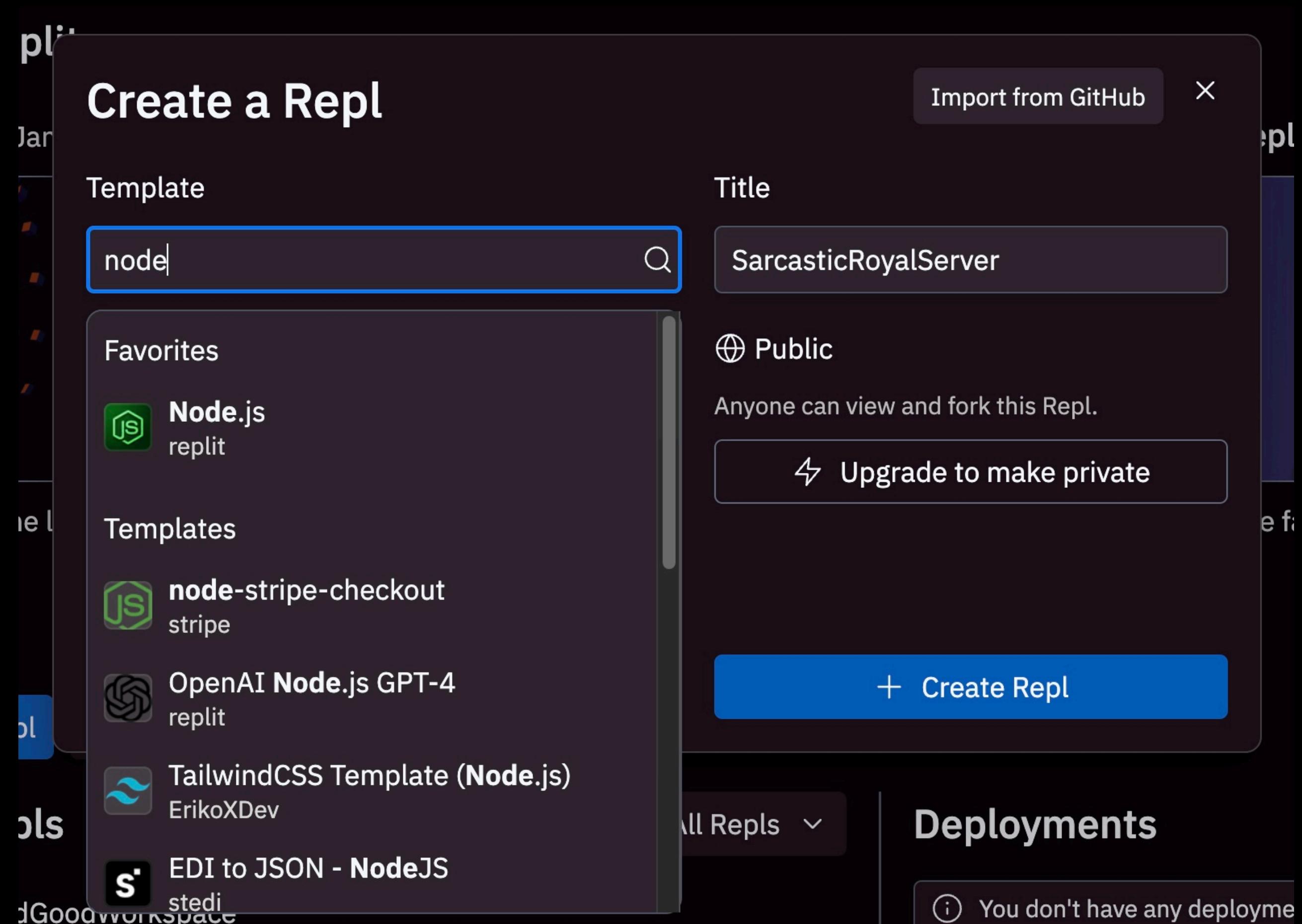
- ✓ Build, test, and deploy directly from the browser
- ✓ Collaborate in real-time with Multiplayer
- ✓ Harness the power of Replit's AI to boost your productivity
- ✓ Join a community of 20 million builders

The main content area has a dark blue background. It features the heading "Create a Replit account" and three social login buttons: "Continue with Google" (with a G icon), "Continue with GitHub" (with a GitHub icon), and "Continue with email →". Below these buttons is a link "Already have an account? Log in". Underneath that, a note states: "By continuing, you agree to Replit's [Terms of Service](#) and [Privacy Policy](#)". At the bottom, a reCAPTCHA notice reads: "This site is protected by reCAPTCHA Enterprise and the Google [Privacy Policy](#) and [Terms of Service](#) apply."

- 1. Authentication (Login with google)**
- 2. Extremely good UI**

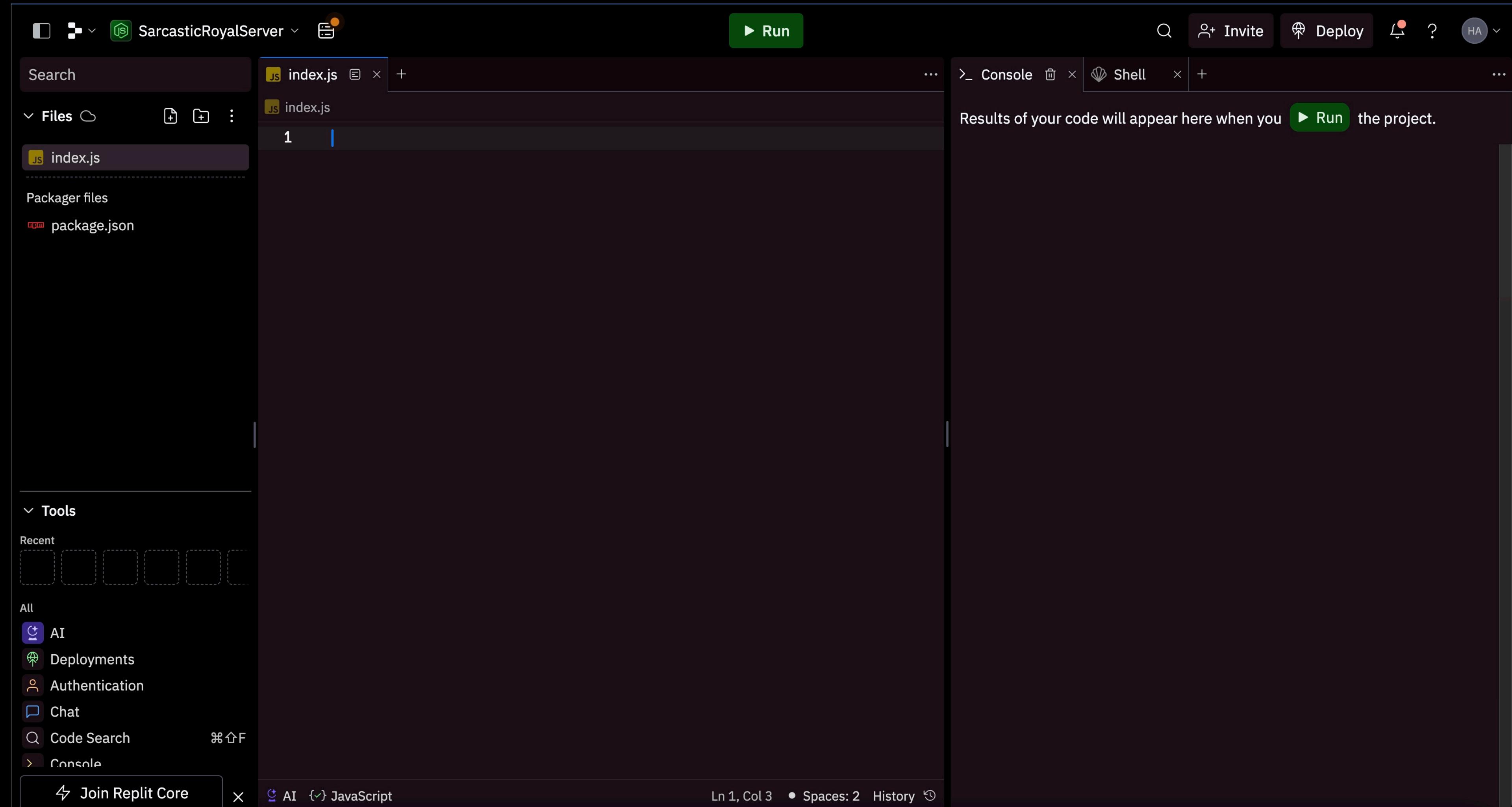
Requirements

User should be able to start a new environment in a selected stack



Requirements

User should be taken to a basic boilerplate repl for that environment



Requirements

User should be able to edit and save their code somewhere

The screenshot shows the Replit IDE interface. On the left, the file tree shows a project named "SarcasticRoyalServer" with files "index.js" and "package.json". The main editor window contains the following code:

```
1 const express = require("express");
2 const app = express();
3
4 app.listen(3000)
```

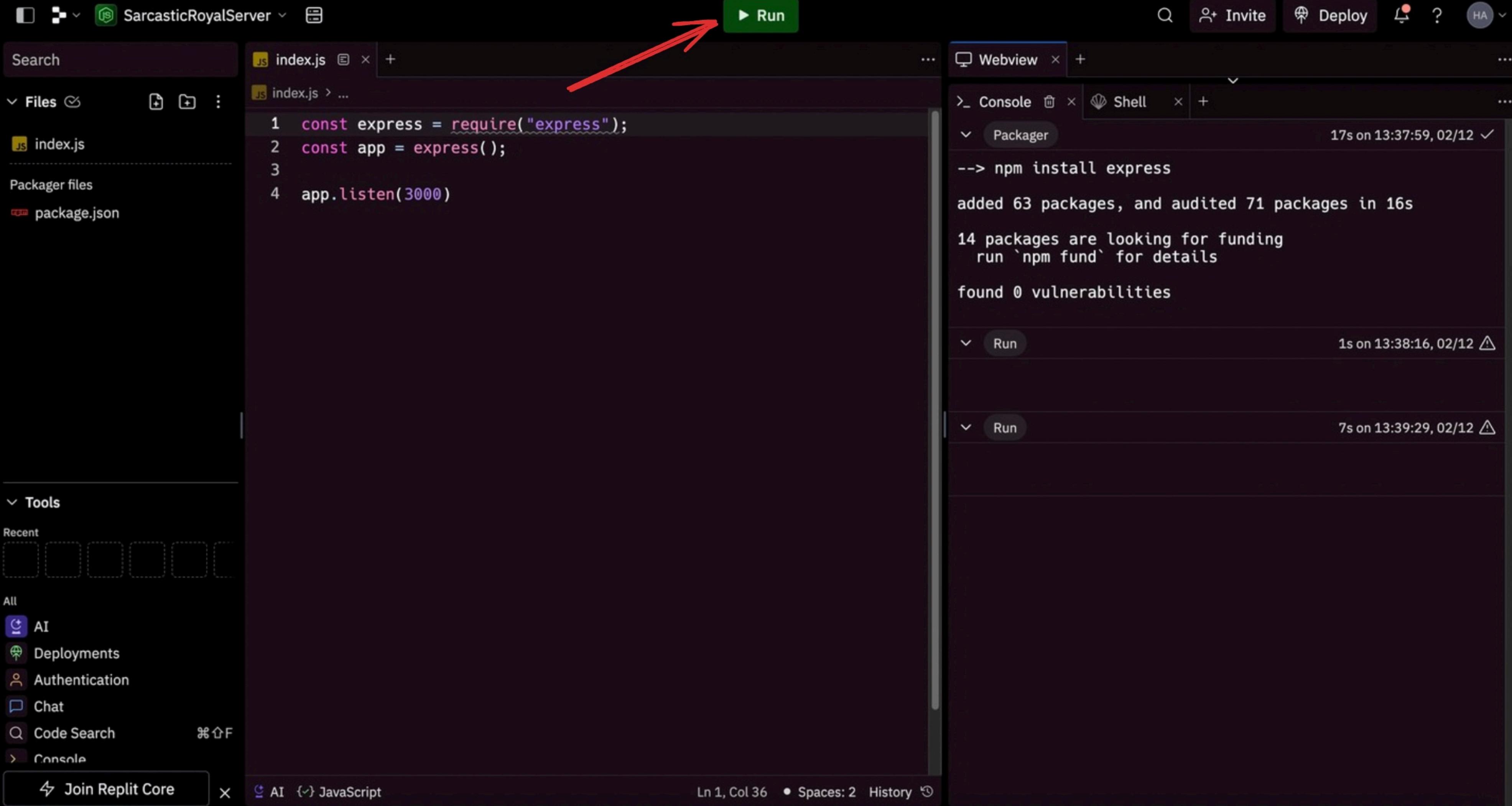
On the right, the terminal window shows the output of running the application:

```
17s on 13:37:59, 02/12 ✓
--> npm install express
added 63 packages, and audited 71 packages in 16s
14 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

The bottom status bar indicates "Ln 1, Col 36 • Spaces: 2 History ⌂".

Requirements

User should be able to run code (both long running and short running)



A screenshot of the Replit IDE interface. On the left, the file tree shows a project named "SarcasticRoyalServer" containing "index.js" and "package.json". The "index.js" file contains the following code:

```
1 const express = require("express");
2 const app = express();
3
4 app.listen(3000)
```

A red arrow points from the text "User should be able to run code (both long running and short running)" to the green "Run" button located at the top right of the editor area. The status bar at the bottom indicates "Ln 1, Col 36 • Spaces: 2 History".

The right side of the interface shows the "Webview" tab, which includes a "Console" tab displaying the output of an npm install command for "express". The output shows:

```
--> npm install express
added 63 packages, and audited 71 packages in 16s
14 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Below the console, there are two "Run" buttons with execution times of "1s" and "7s" respectively.

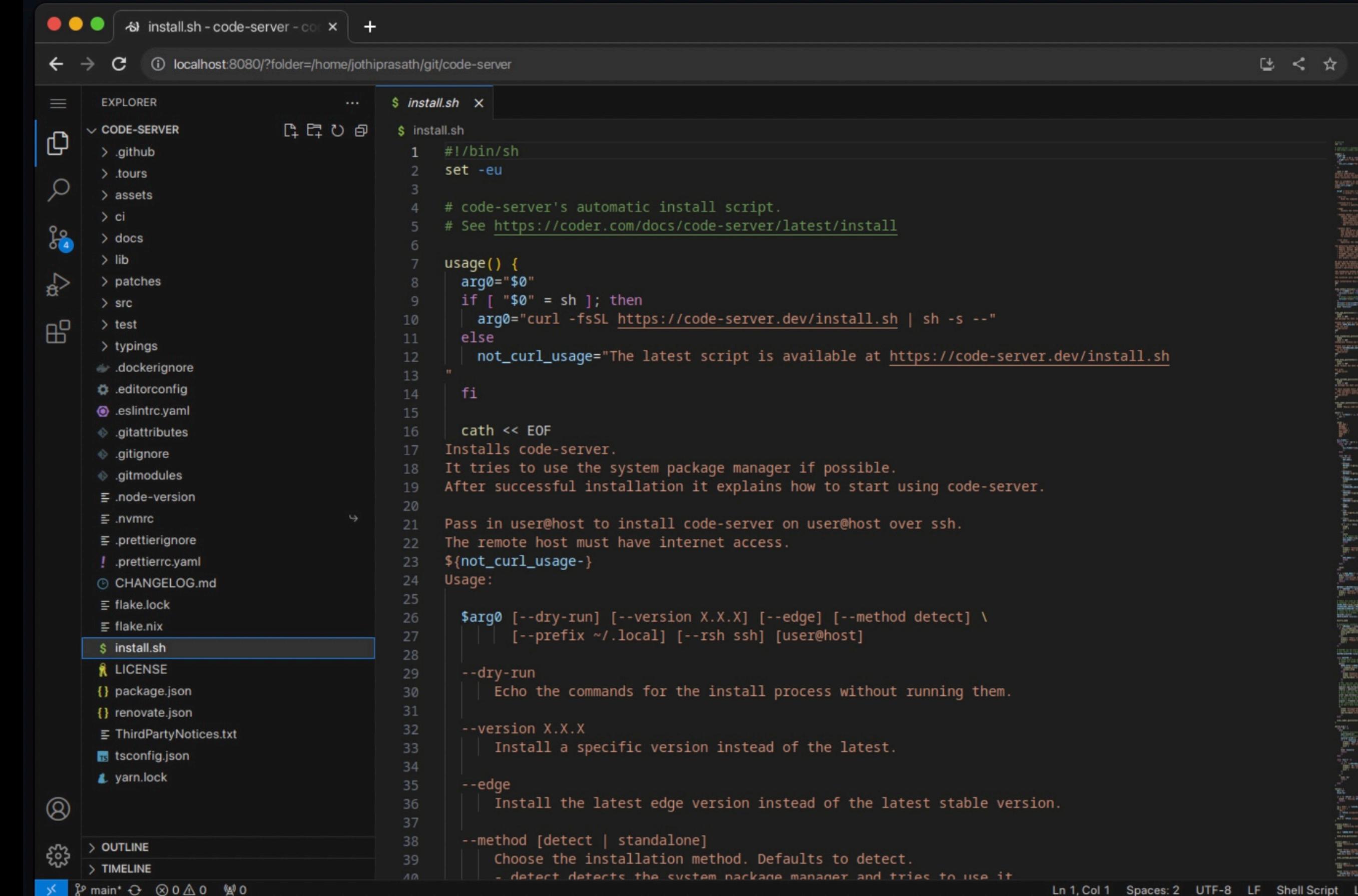
We'll be taking 3 approaches to solve this problem

Callout -

You can use an external service/codebase that does a lot of this for you.

If you're a startup, you probably want to pick one of these services and not build this from scratch

<https://github.com/coder/code-server>
(Github codespaces)



```
#!/bin/sh
set -eu

# code-server's automatic install script.
# See https://coder.com/docs/code-server/latest/install

usage() {
    arg0="$0"
    if [ "$0" = sh ]; then
        arg0="curl -fsSL https://code-server.dev/install.sh | sh -s --"
    else
        not(curl_usage="The latest script is available at https://code-server.dev/install.sh")
    fi
    cat << EOF
Installs code-server.
It tries to use the system package manager if possible.
After successful installation it explains how to start using code-server.
Pass in user@host to install code-server on user@host over ssh.
The remote host must have internet access.
${not(curl_usage)-}
Usage:

$arg0 [--dry-run] [--version X.X.X] [--edge] [--method detect] \
       [--prefix ~/.local] [--rsh ssh] [user@host]
--dry-run
    Echo the commands for the install process without running them.
--version X.X.X
    Install a specific version instead of the latest.
--edge
    Install the latest edge version instead of the latest stable version.
--method [detect | standalone]
    Choose the installation method. Defaults to detect.
    - detect detects the system package manager and tries to use it
EOF
}

not(curl_usage="The latest script is available at https://code-server.dev/install.sh")

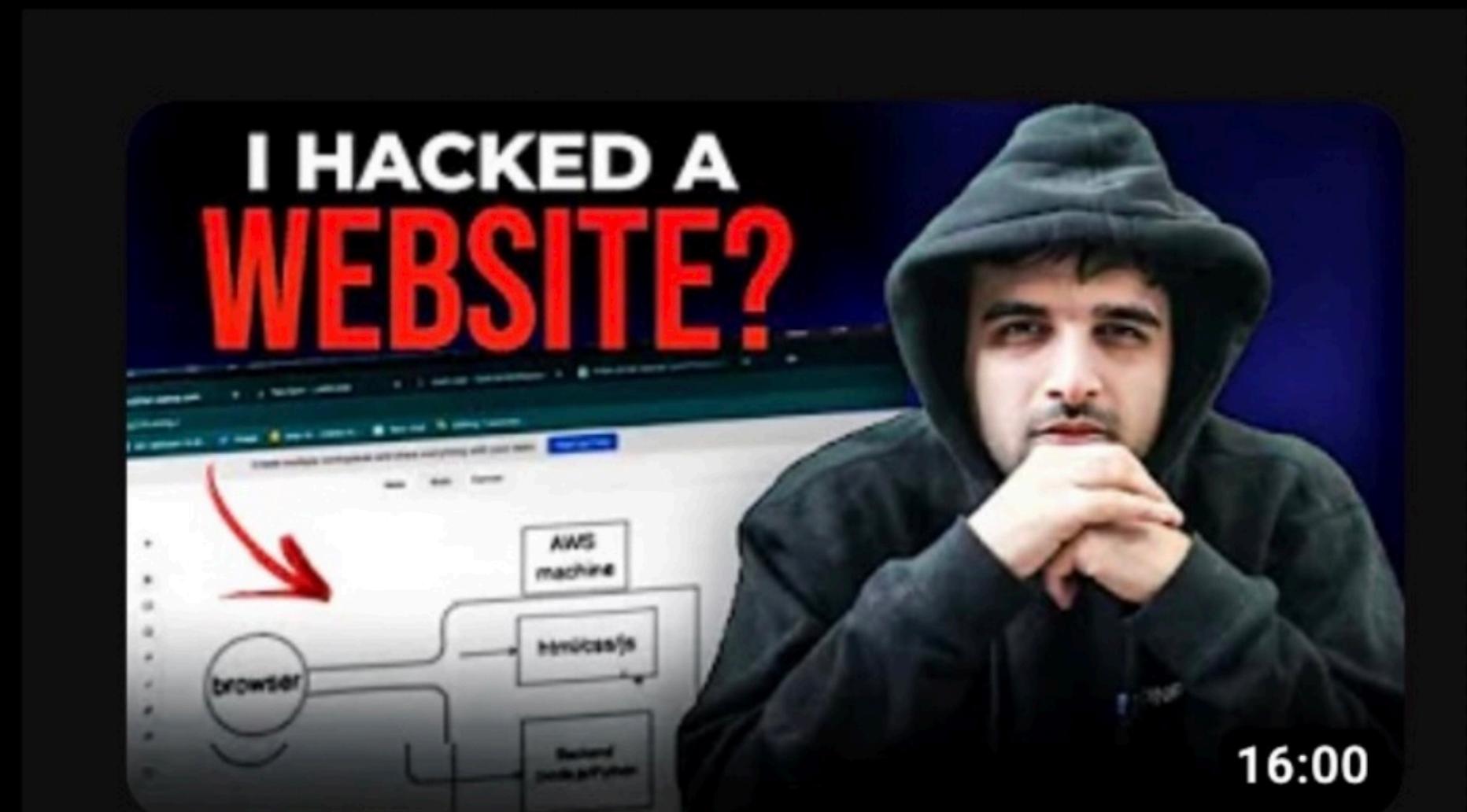
$arg0=$0
if [ "$0" = sh ]; then
    arg0="curl -fsSL https://code-server.dev/install.sh | sh -s --"
else
    not(curl_usage="The latest script is available at https://code-server.dev/install.sh")
fi

cat << EOF
Installs code-server.
It tries to use the system package manager if possible.
After successful installation it explains how to start using code-server.
Pass in user@host to install code-server on user@host over ssh.
The remote host must have internet access.
${not(curl_usage)-}
Usage:

$arg0 [--dry-run] [--version X.X.X] [--edge] [--method detect] \
       [--prefix ~/.local] [--rsh ssh] [user@host]
--dry-run
    Echo the commands for the install process without running them.
--version X.X.X
    Install a specific version instead of the latest.
--edge
    Install the latest edge version instead of the latest stable version.
--method [detect | standalone]
    Choose the installation method. Defaults to detect.
    - detect detects the system package manager and tries to use it
EOF
```

Beginner friendly

Great way to build intuition on how you can build something like this
DO NOT use it in production/in an interview



Biggest Mistakes Website Developers Make
in 2023

38K views • 8 months ago

Why you shouldn't do it

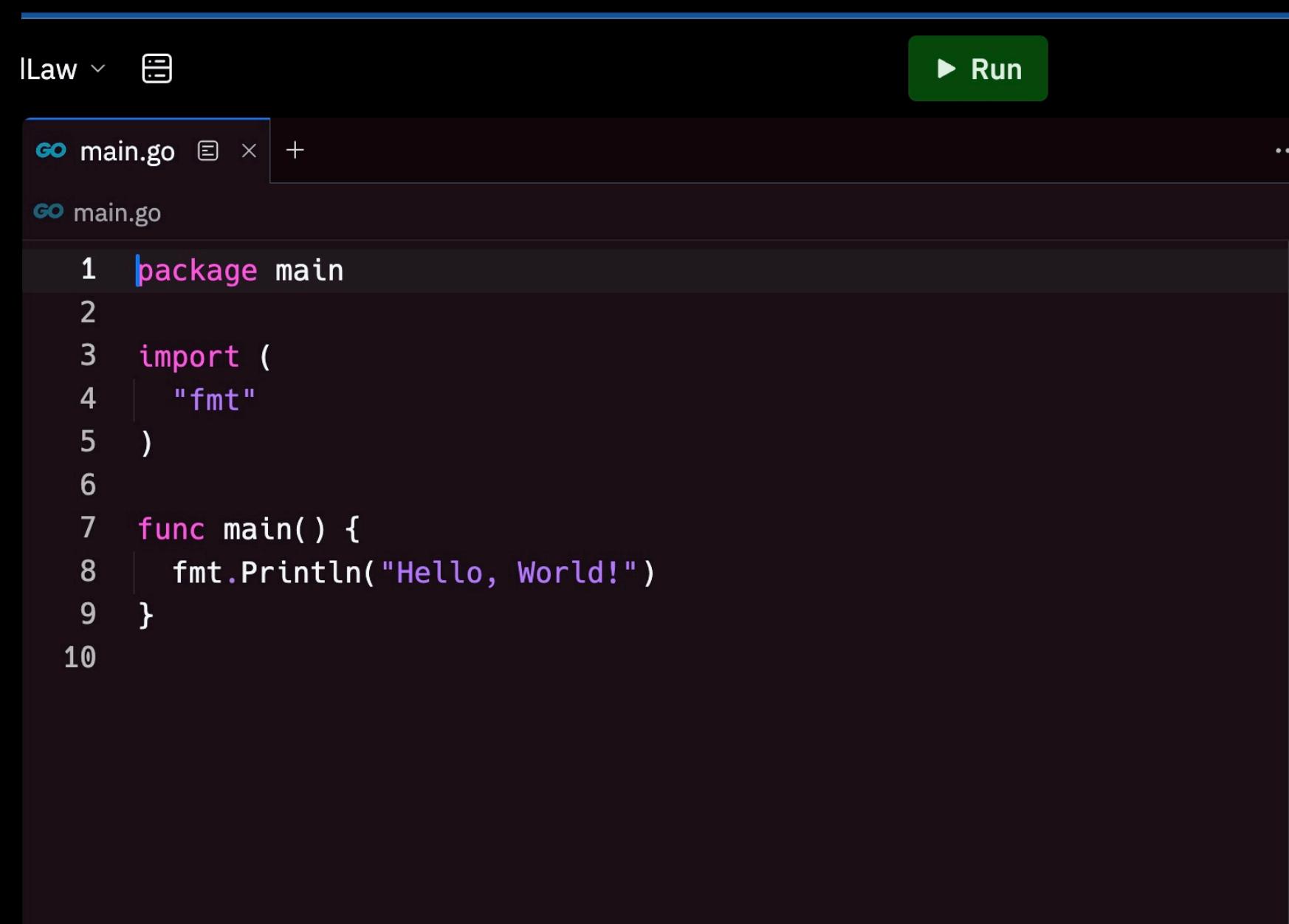
Downsides

1. Insecure Remote code execution
2. Single server setup, doesn't autoscale
3. Port conflicts between two users (every user is sharing resources on the same server)
4. Terminal is **extremely** rudimentary
5. Very ugly package management

Why is building repl.it hard?

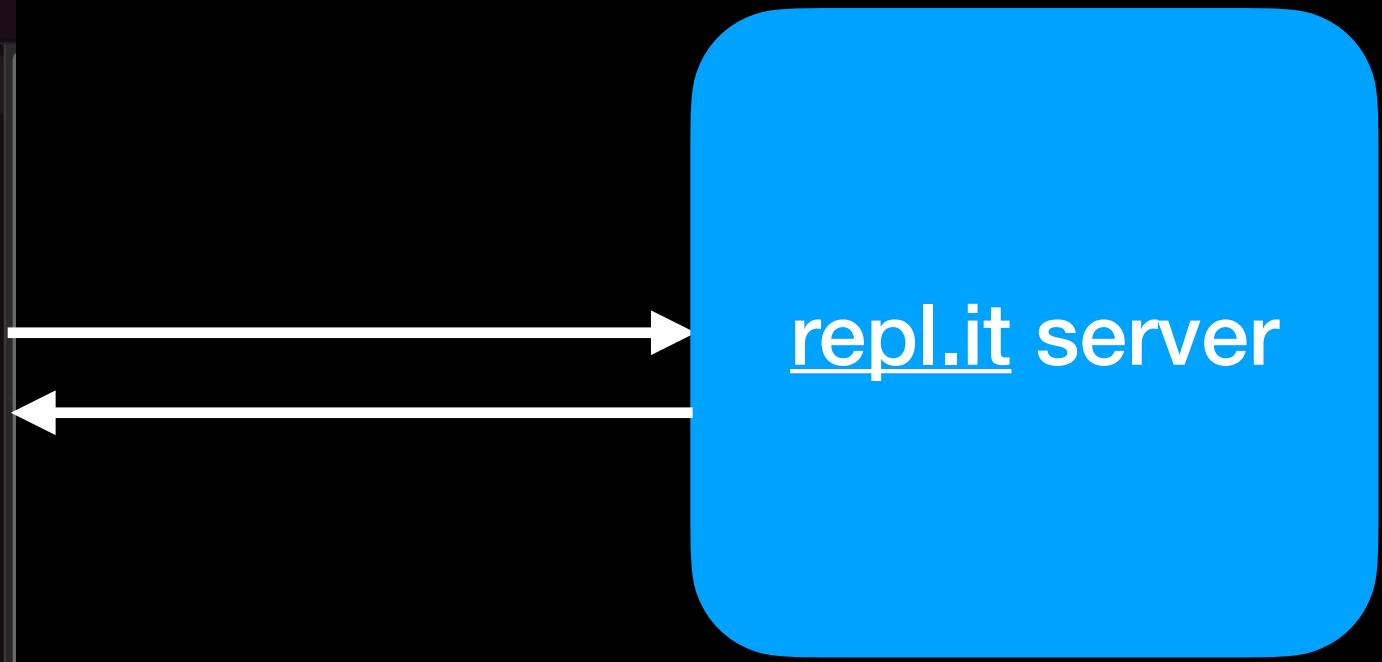
Why is building repl.it hard?

1. Remote code execution



A screenshot of the repl.it web interface. At the top, there's a header with a dropdown menu labeled "ILaw" and a "Run" button. Below the header is a code editor window titled "main.go". The code editor contains the following Go code:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, World!")
9 }
10
```



Your browser can't run C++/Rust/Go/React (Not technically true)

Why is building repl.it hard?

You need to give user VCPU guarantees

1. Remote code execution
2. Long running processes

The screenshot shows the Replit IDE interface. On the left, the file tree displays files like README.md, App.css, App.jsx, index.jsx, index.html, README.md, tsconfig.json, vite.config.js, package.json, and packager files. The central editor window shows the following code in App.jsx:

```
1 import './App.css'
2
3 export default function App() {
4     return (
5         <main>
6             | React * + Vite ⚡ + Replit
7         </main>
8     )
9 }
```

To the right, the "Webview" tab shows a browser window with the URL "/". A blue bar at the top of the browser says "Ready to share with the world? Deploy your app". Below the browser is a "Run on Replit" button. At the bottom, the "Console" tab shows the command "react-javascript@1.0.0 dev" and "vite". The output shows "VITE v5.0.0 ready in 4864 ms" and "Local: http://localhost:5173/ Network: http://172.31.196.17:5173/ press h + enter to show help".

Example of short running process - Leetcode

Why is building repl.it hard?

1. Remote code execution
2. Long running processes
3. Shell access inside browser

The screenshot shows a development environment with several panels:

- Code Editor:** Shows `App.jsx` with React code.
- Webview:** Shows a browser window with the message "Ready to share with the world? Deploy your app".
- Console:** Shows a terminal window with the command `ls` run in a directory named `FlashyBrokenRobots`. The output includes `index.html`, `package-lock.json`, `src`, `node_modules`, `public`, `tsconfig.json`, `package.json`, `README.md`, and `vite.config.js`. This panel is highlighted with a red rounded rectangle.

At the bottom, status bars indicate "AI {v} JavaScript React" and "Ln 1, Col 1 • Spaces: 2 History".

```
1 import './App.css'
2 
3 export default function App() {
4   return (
5     <main>
6       React * + Vite < + Replit
7     </main>
8   )
9 }
10
```

```
~/FlashyBrokenRobots$ ls
index.html    package-lock.json    src
node_modules  public              tsconfig.json
package.json   README.md        vite.config.js
~/FlashyBrokenRobots$
```

Why is building repl.it hard?

1. Remote code execution
2. Long running processes
3. Shell access inside browser
4. File storage (not difficult)

The screenshot shows the repl.it development environment. On the left, the file tree displays a project structure with files like README.md, App.jsx, index.js, and package.json. A red box highlights the package.json file. The central area shows the code editor with the following content:

```
1 import './App.css'
2
3 export default function App() {
4   return (
5     <main>
6       React * + Vite > + Replit
7     </main>
8   )
9 }
10
```

On the right, there's a preview window showing a webview of the application, a deployment status message "Ready to share with the world? Deploy your app", and a terminal window showing the command `ls` output:

```
~/FlashyBrokenRobots$ ls
index.html  package-lock.json  src
node_modules  public          tsconfig.json
package.json  README.md      vite.config.js
~/FlashyBrokenRobots$
```

A "Run on Replit" button is visible in the bottom right of the terminal area.

Example of not requiring file storage - Leetcode

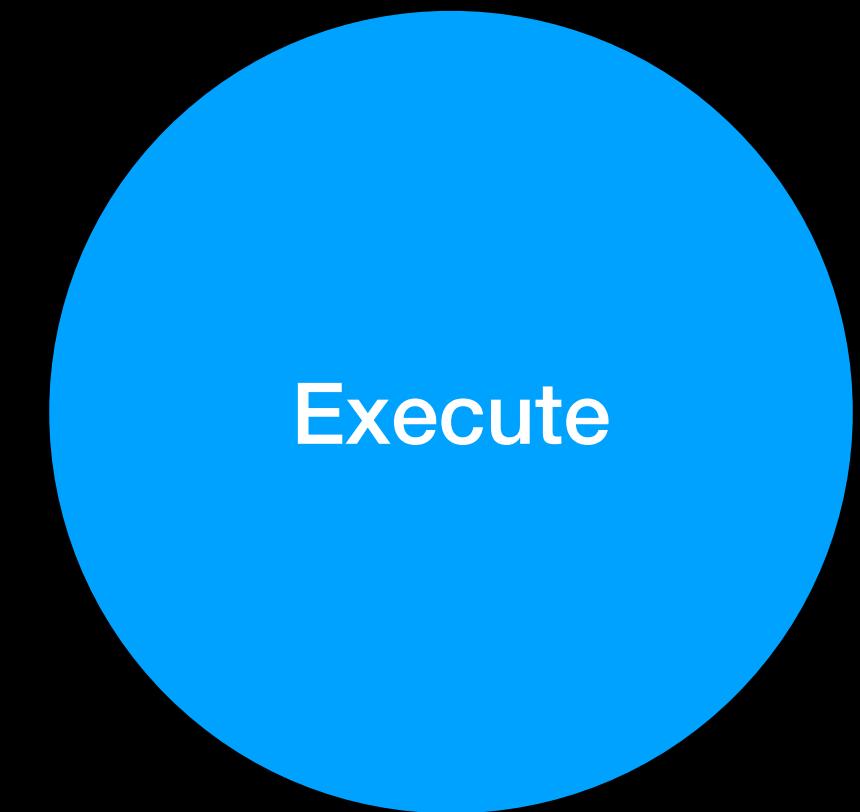
Why is building repl.it hard?

Let's see how we can achieve all of these one by one

1. Remote code execution
2. Long running processes
3. Shell access inside browser
4. File storage (not difficult)

You will have 1 big monolith for this part

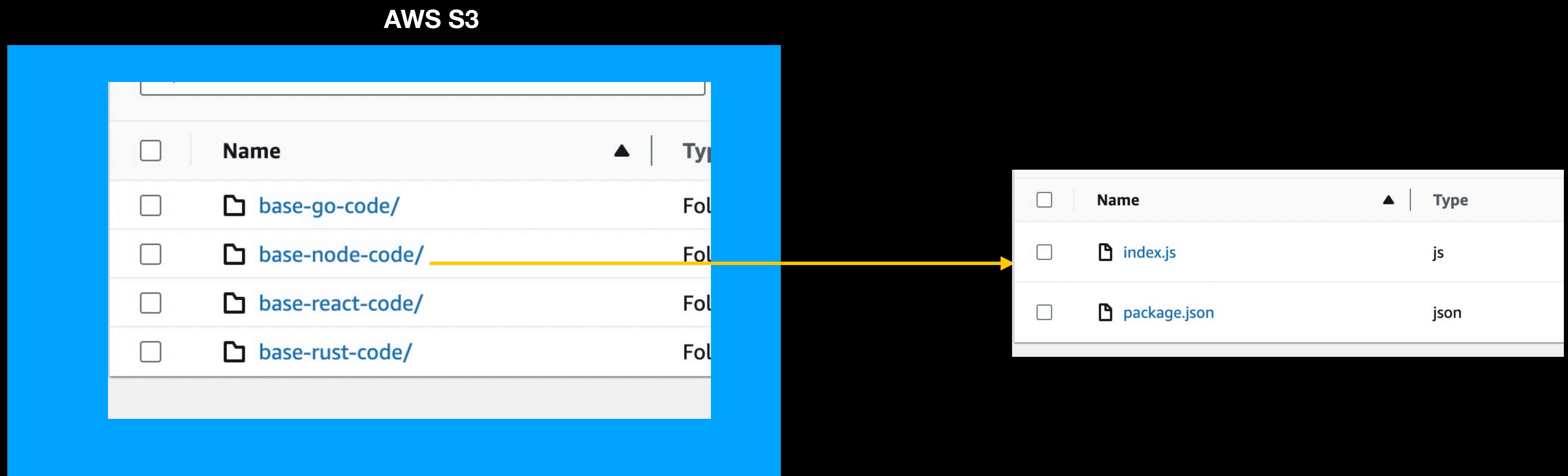
Execution service



- 1. Remote code execution**
- 2. Long running processes**
- 3. Shell access inside browser**
- 4. File storage (not difficult)**

Execution service

Step 0.1 - Keeping a copy of base projects in S3



Execution service

Step 0.2 - Bringing all languages you support (Node, Go, Rust) to this machine

Mac machine/AWS Server (Execute service)

Install node

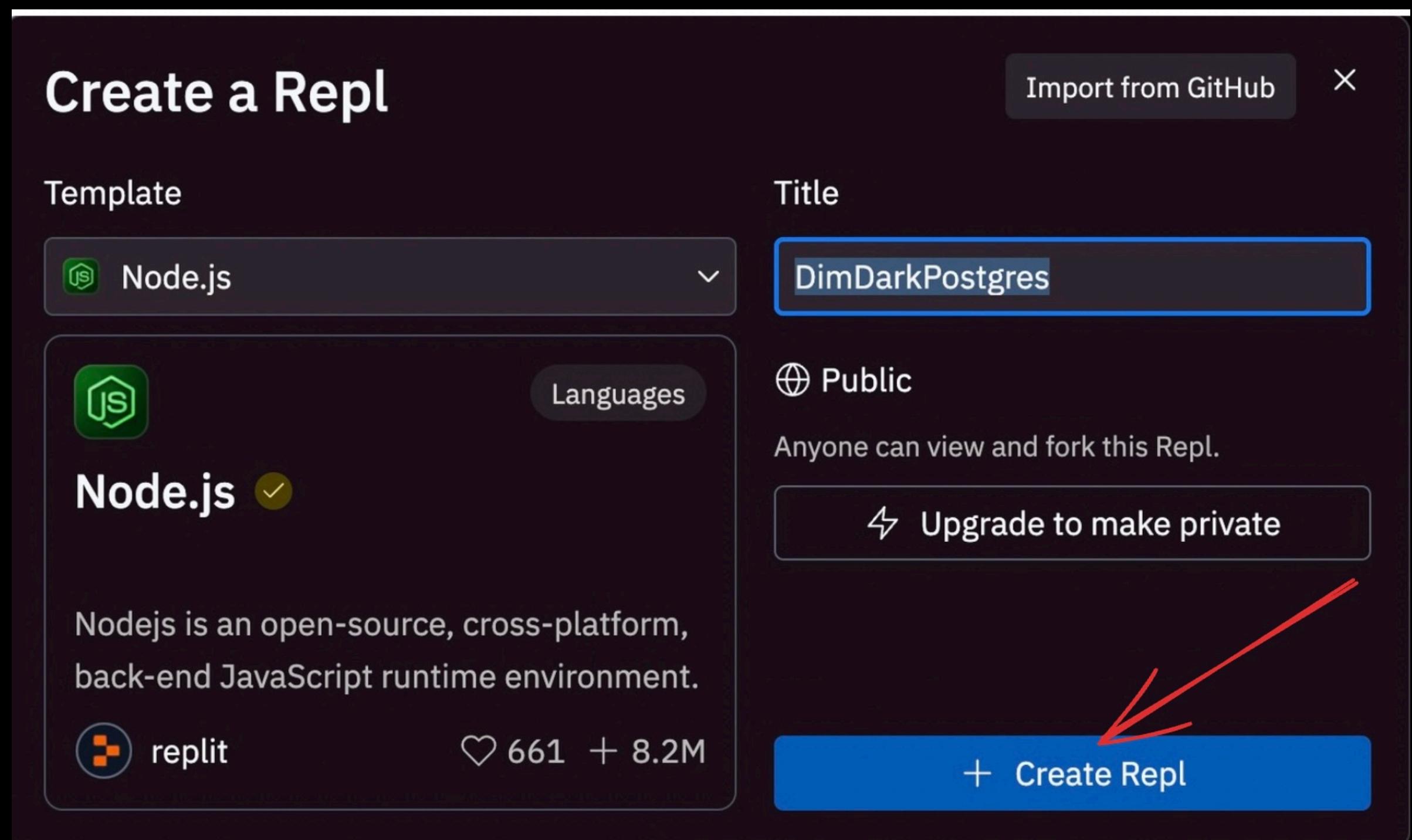
Install rust

Install Golang

...

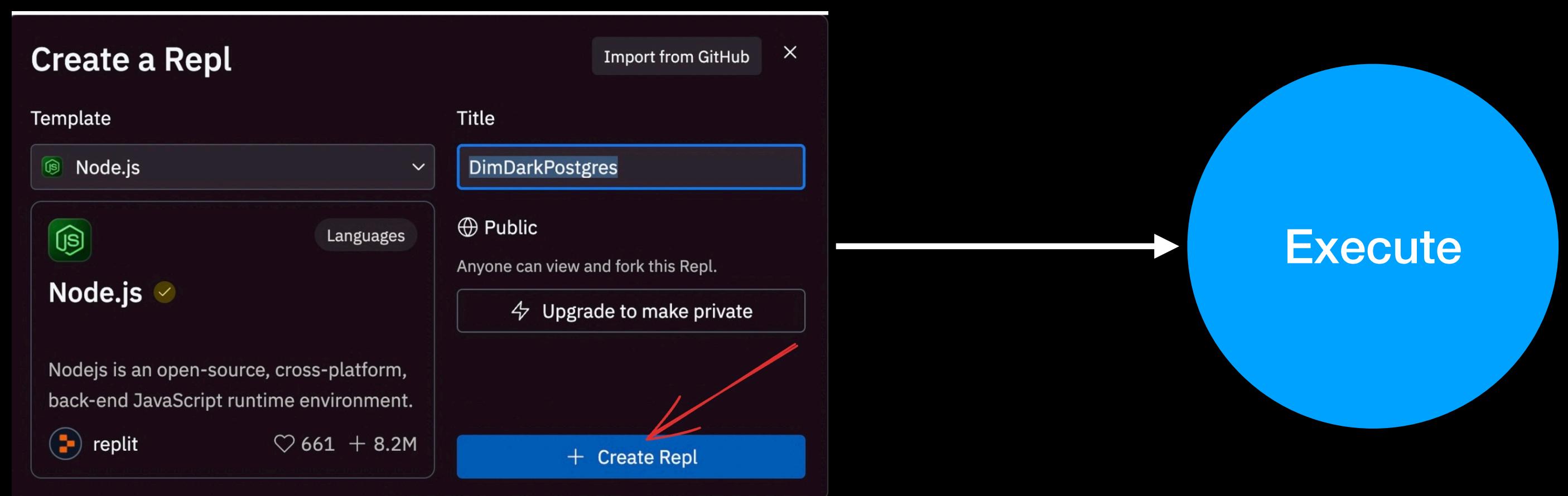
Execution service

Step 1 - Initialising the repl



Execution service

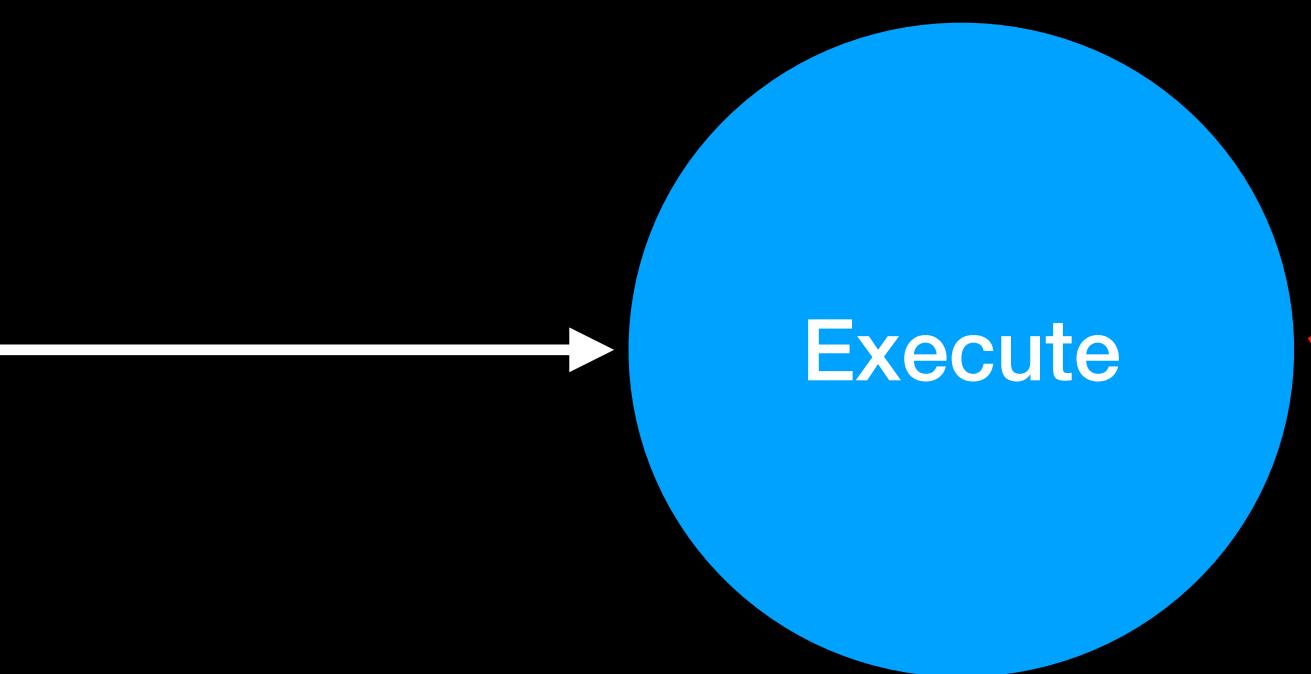
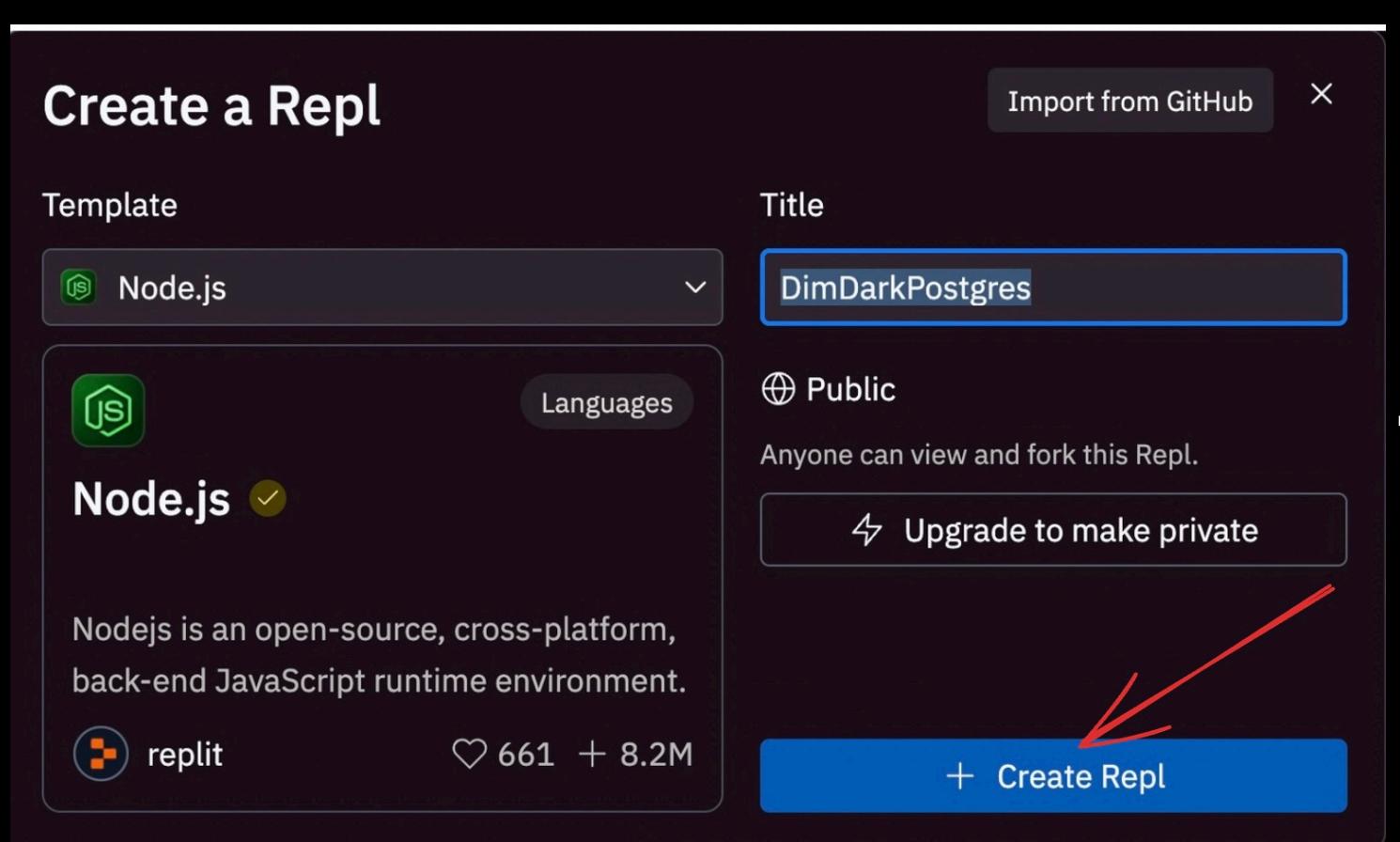
Step 1 - Initialising the repl



Execution service

Step 1 - Initialising the repl

Copy over the base image to
s3://images/{id}



The screenshot shows an S3 bucket named 'DimDarkPostgres/'. Inside the bucket, there are four folders: 'base-go-code/', 'base-node-code/' (highlighted with a red box), 'base-react-code/', and 'base-rust-code/'. In the 'base-node-code/' folder, there are two objects: 'index.js' and 'package.json'. A red arrow points from the 'Create Repl' button in the first screenshot to the 'base-node-code/' folder in the second screenshot.

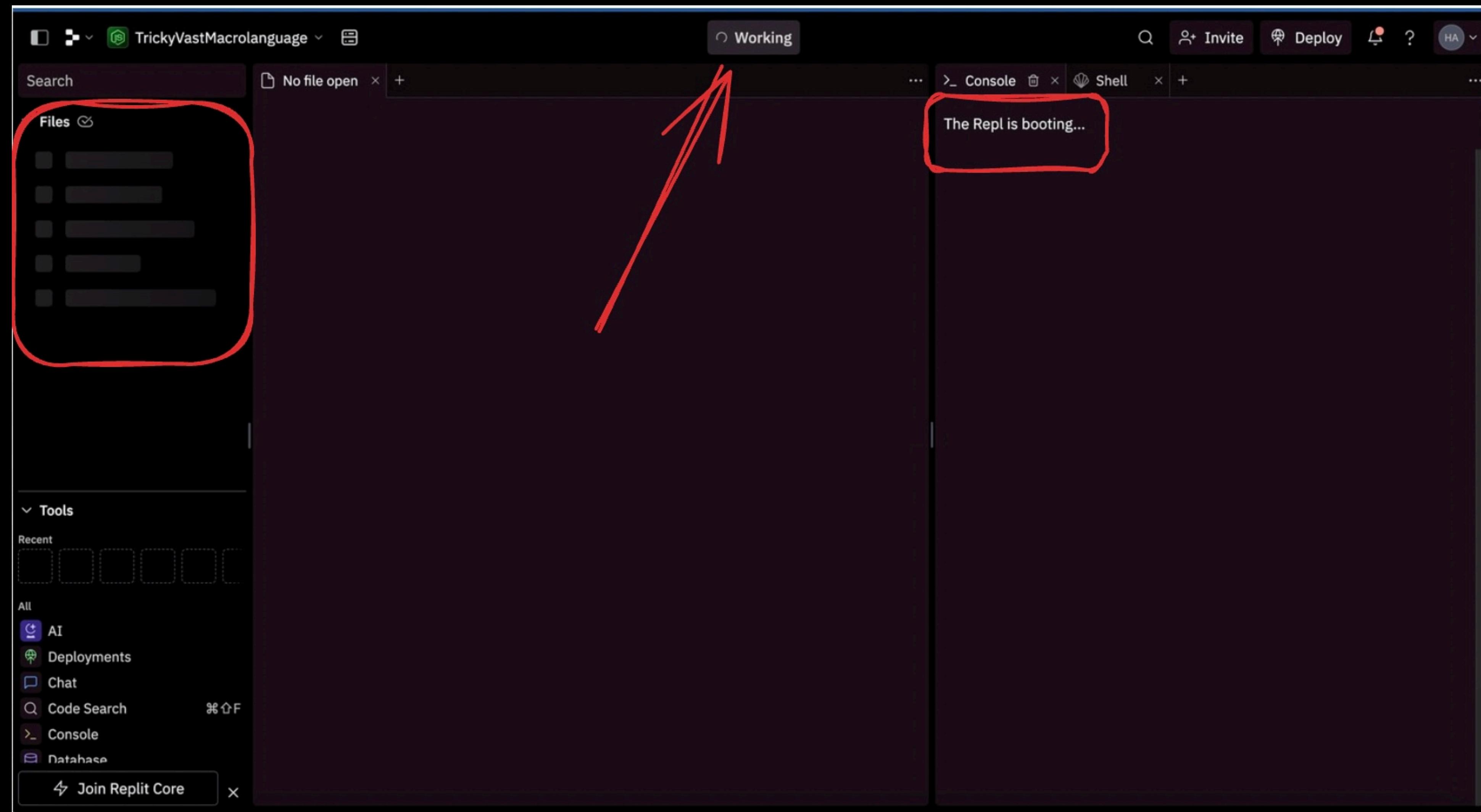
Name	Type
base-go-code/	Folder
base-node-code/	Folder
base-react-code/	Folder
base-rust-code/	Folder

DimDarkPostgres/

Name	Type
index.js	js
package.json	json

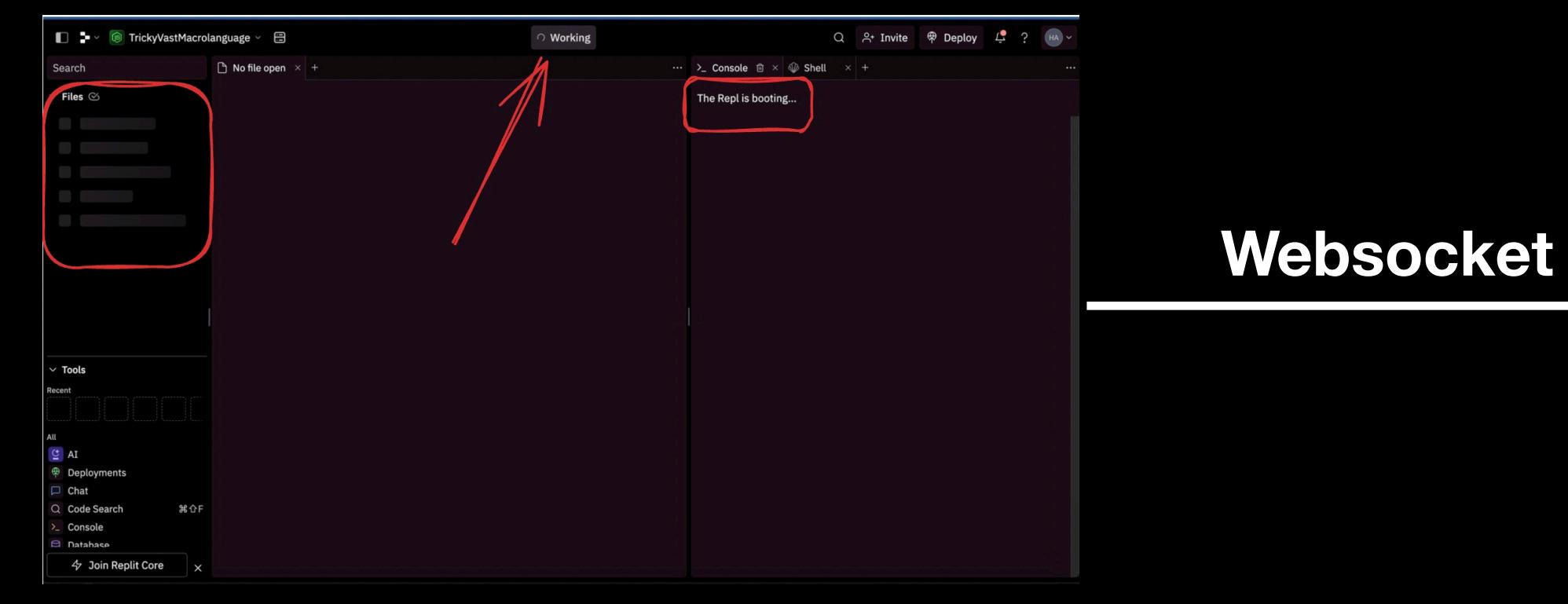
Execution service

Step 2 - Taking the user to the edit screen



Execution service

Step 3 - Initialise a ws connection

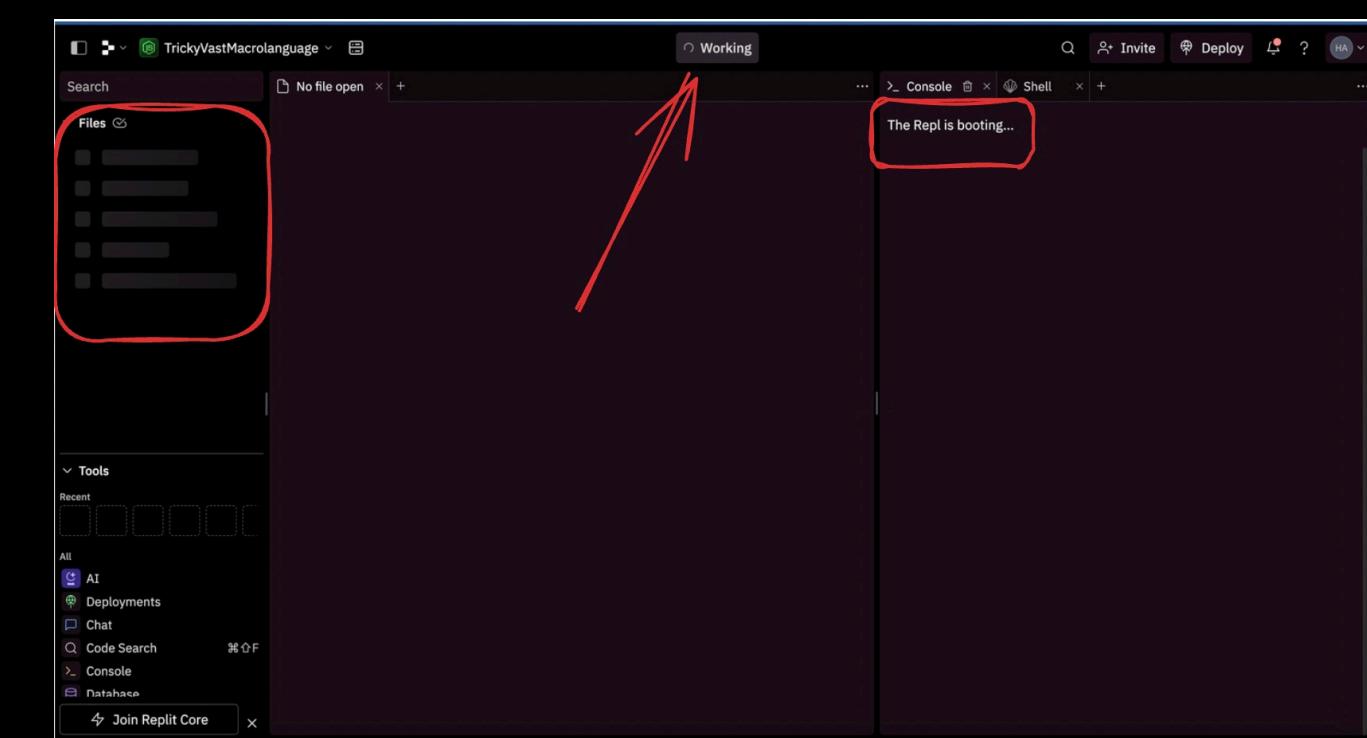


WebSocket

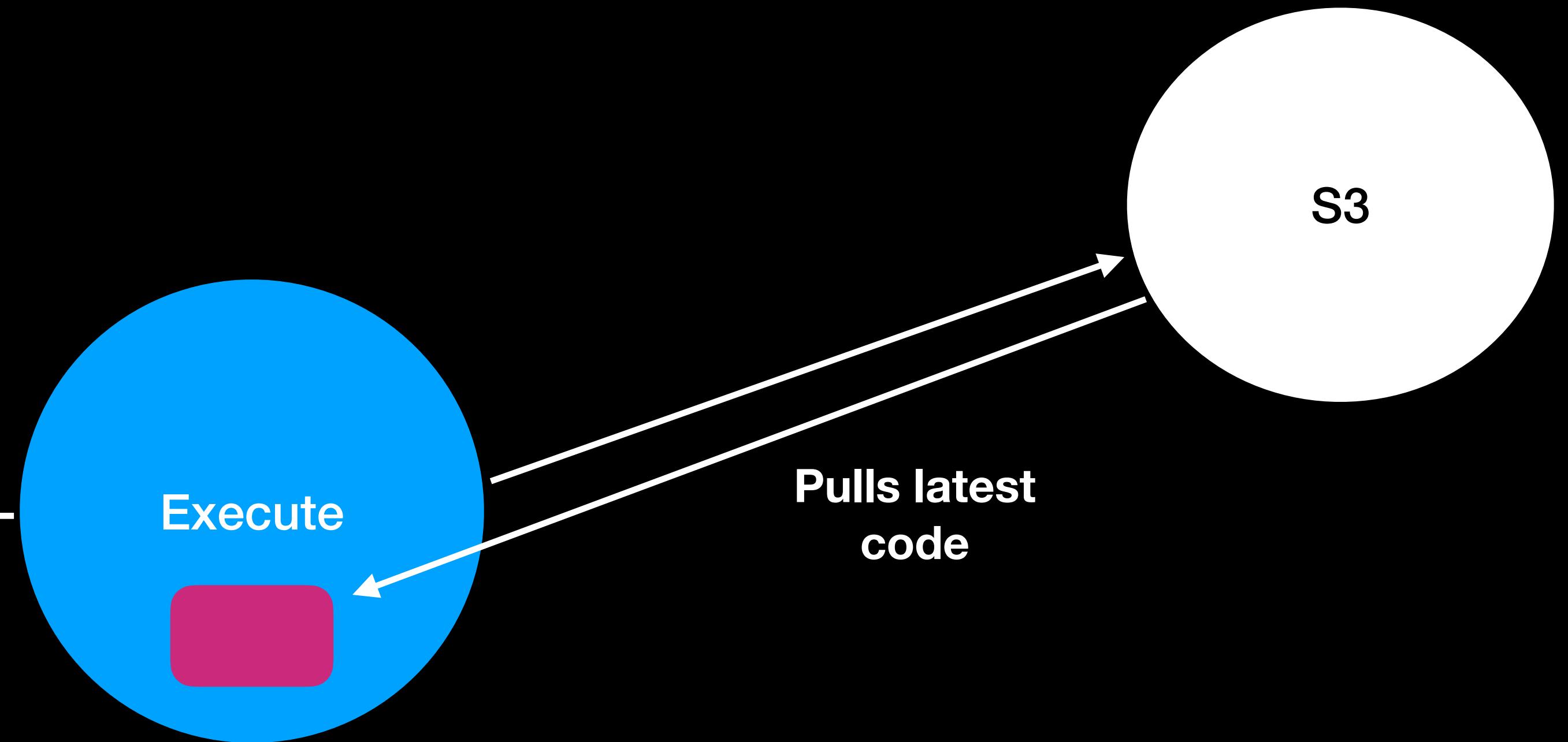
Execute

Execution service

Step 4 -Bring the users code to the VM

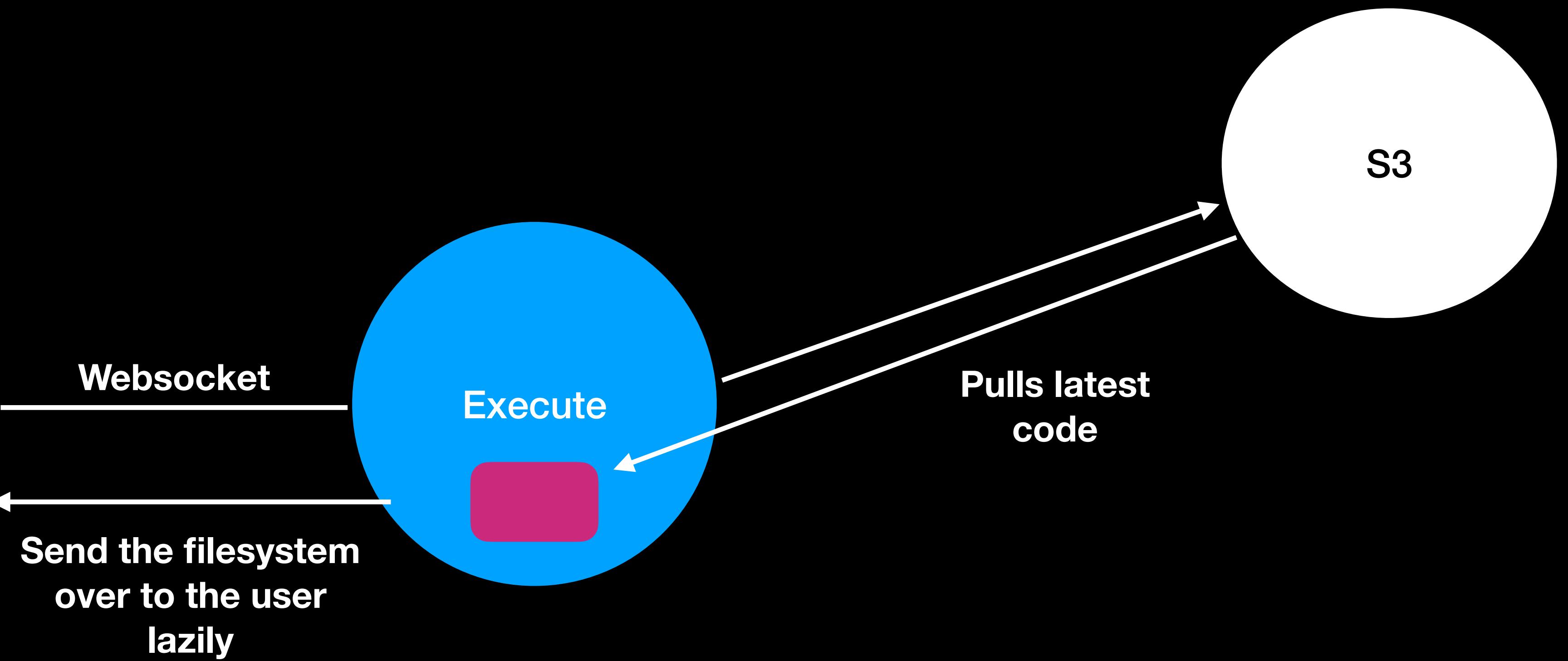
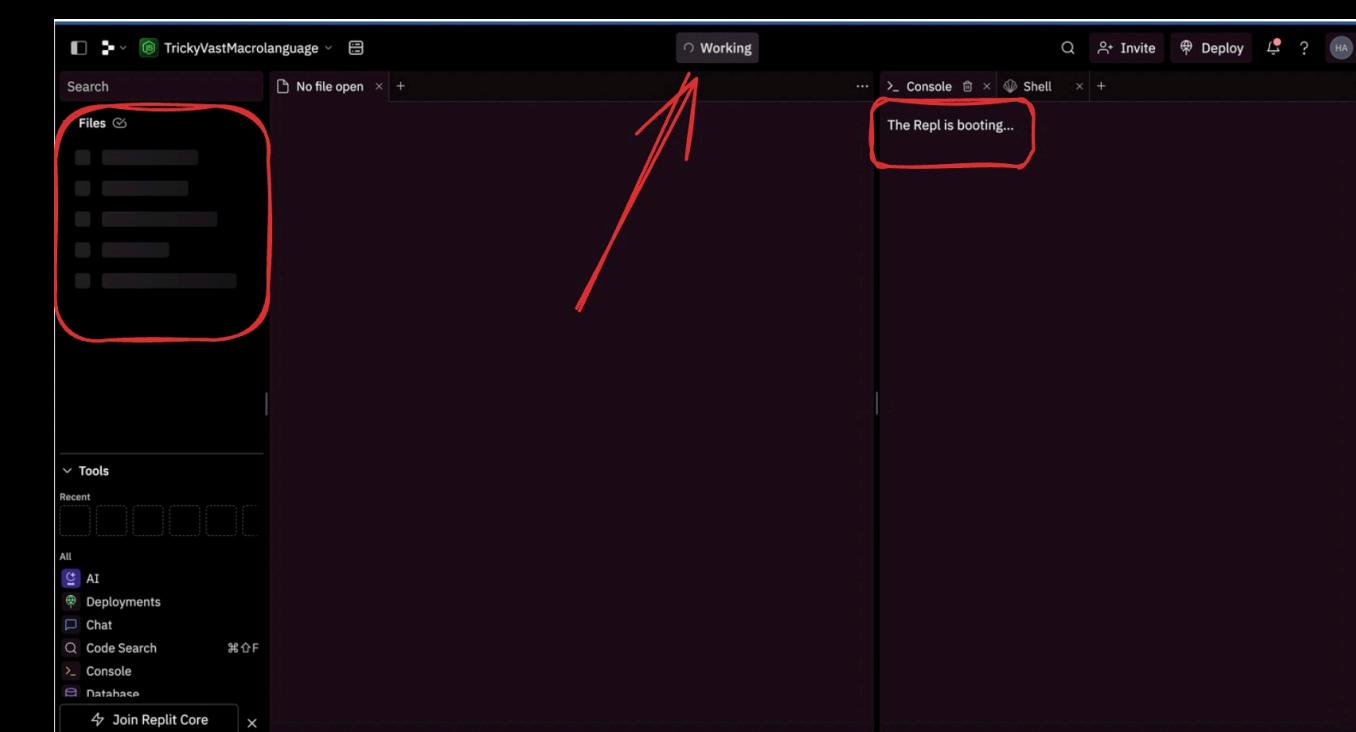


websocket



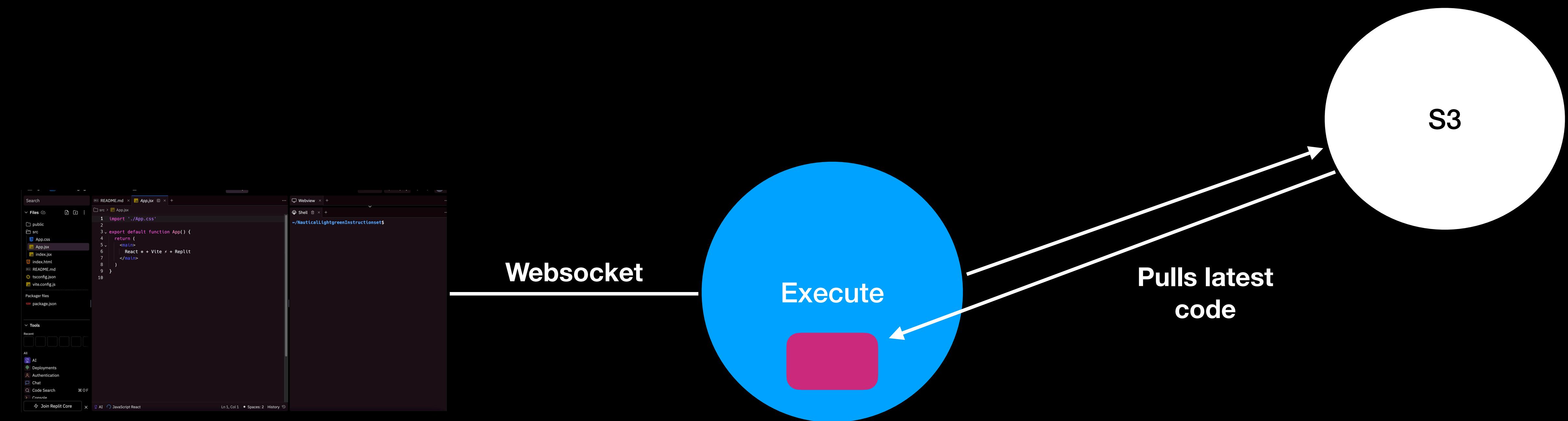
Execution service

Step 4 -Bring the users code to the VM



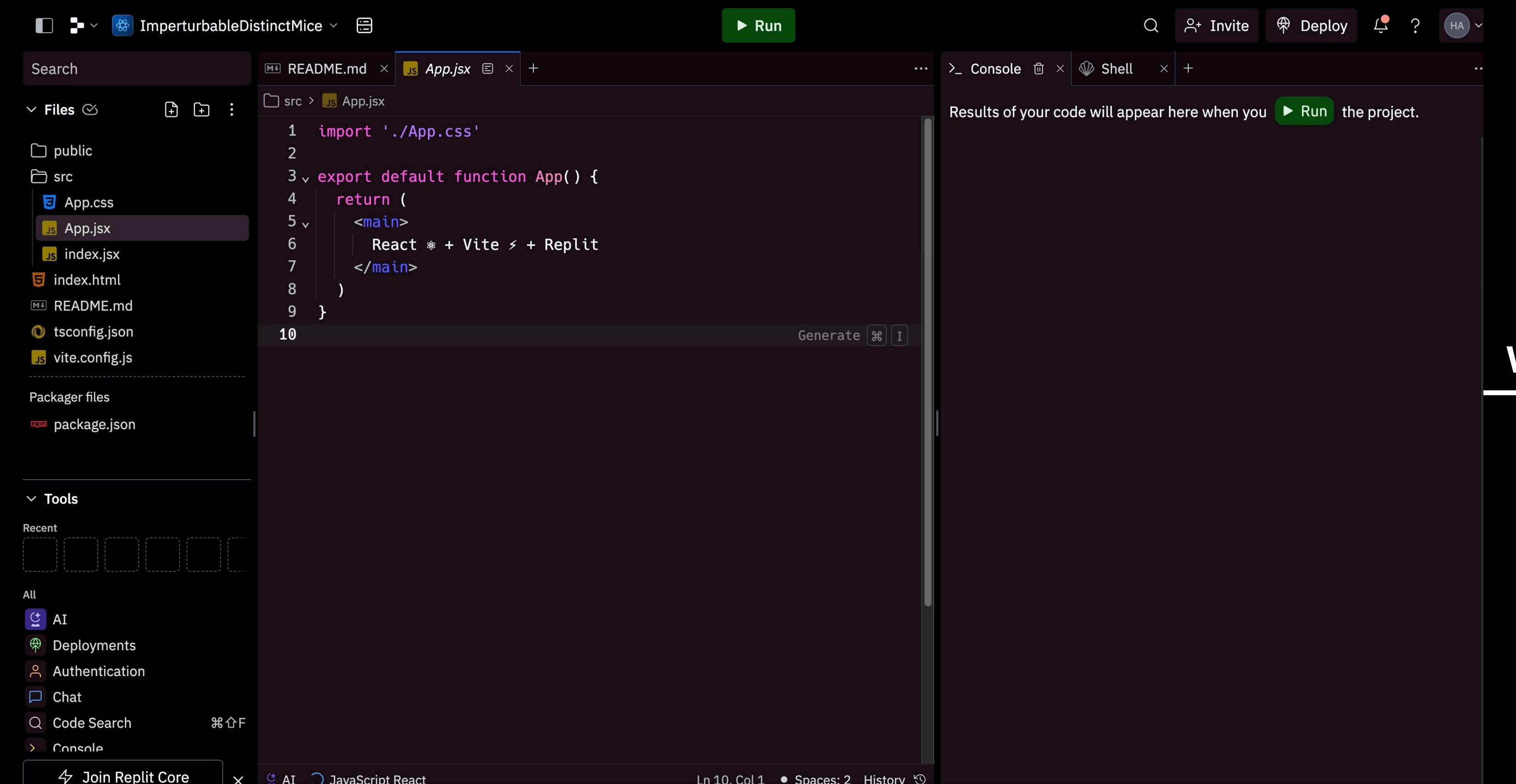
Execution service

Step 5 - Let the user edit files



Execution service

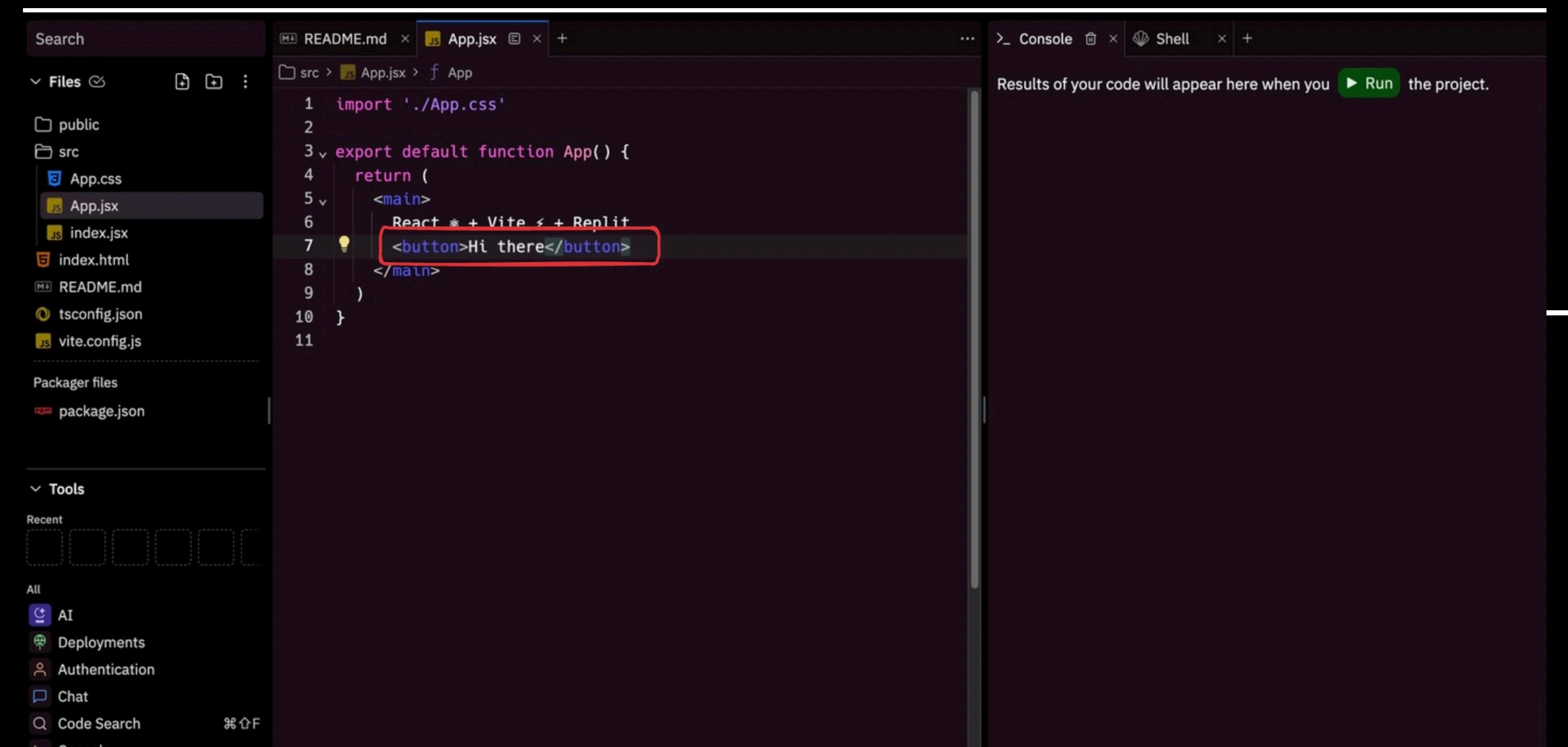
Step 5 - Let the user edit files



A screenshot of the Replit IDE interface. The top bar shows the project name "ImperturbableDistinctMice". The main area displays the code for `App.jsx`:import './App.css'
export default function App() {
 return (
 <main>
 | React * + Vite & + Replit
 </main>
)
}The sidebar on the left shows the file structure: `src` folder containing `App.css`, `App.jsx` (selected), `index.jsx`, and `index.html`. Other files like `README.md`, `tsconfig.json`, and `vite.config.js` are also listed. The bottom right corner of the interface features a large blue circle with the word "Websocket" at the top and "Execute" below it, with a red square button in the center.

Execution service

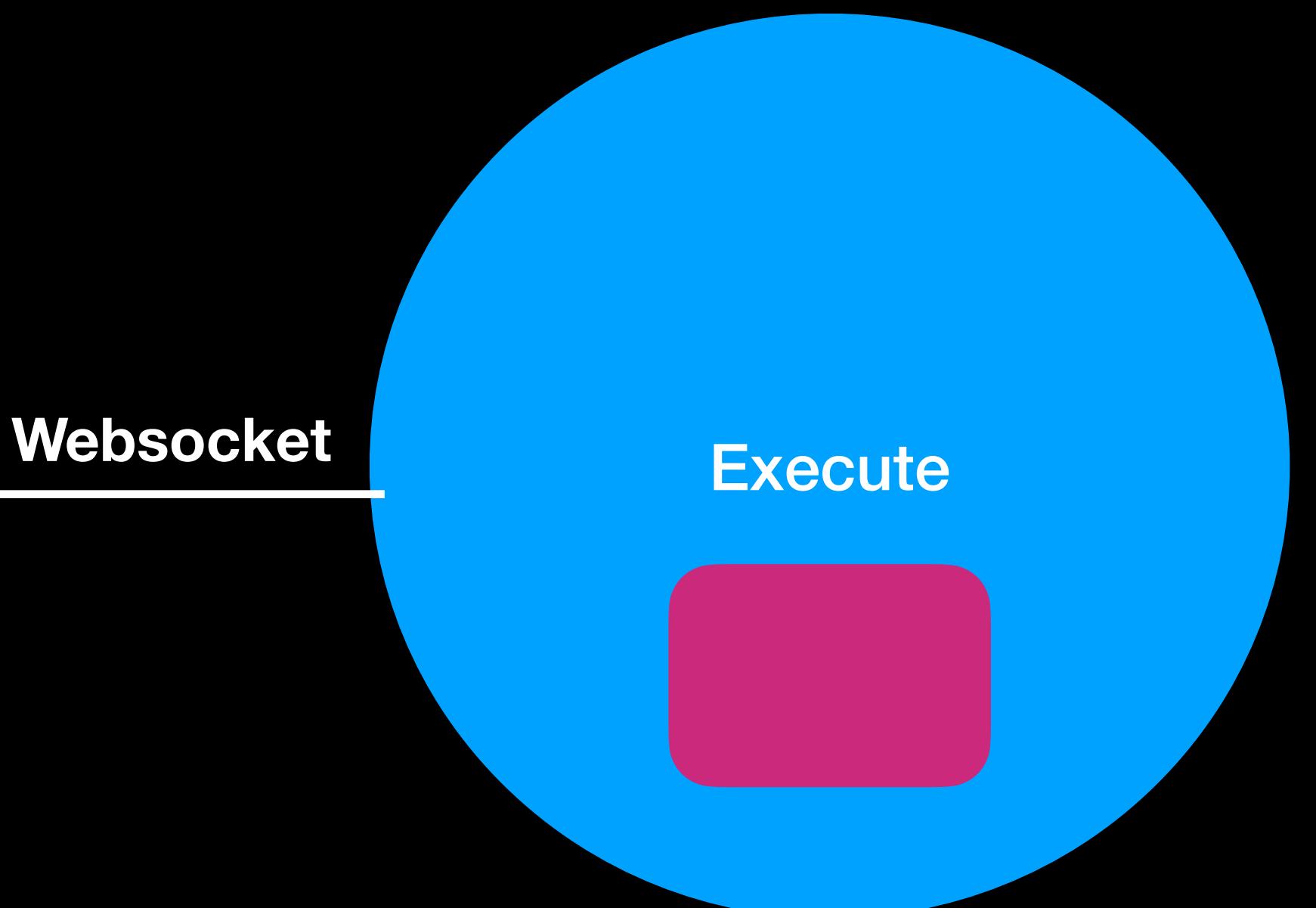
Step 5 - Let the user edit files



A screenshot of a code editor interface, likely Replit, showing a file named `App.jsx`. The code contains a simple React component:

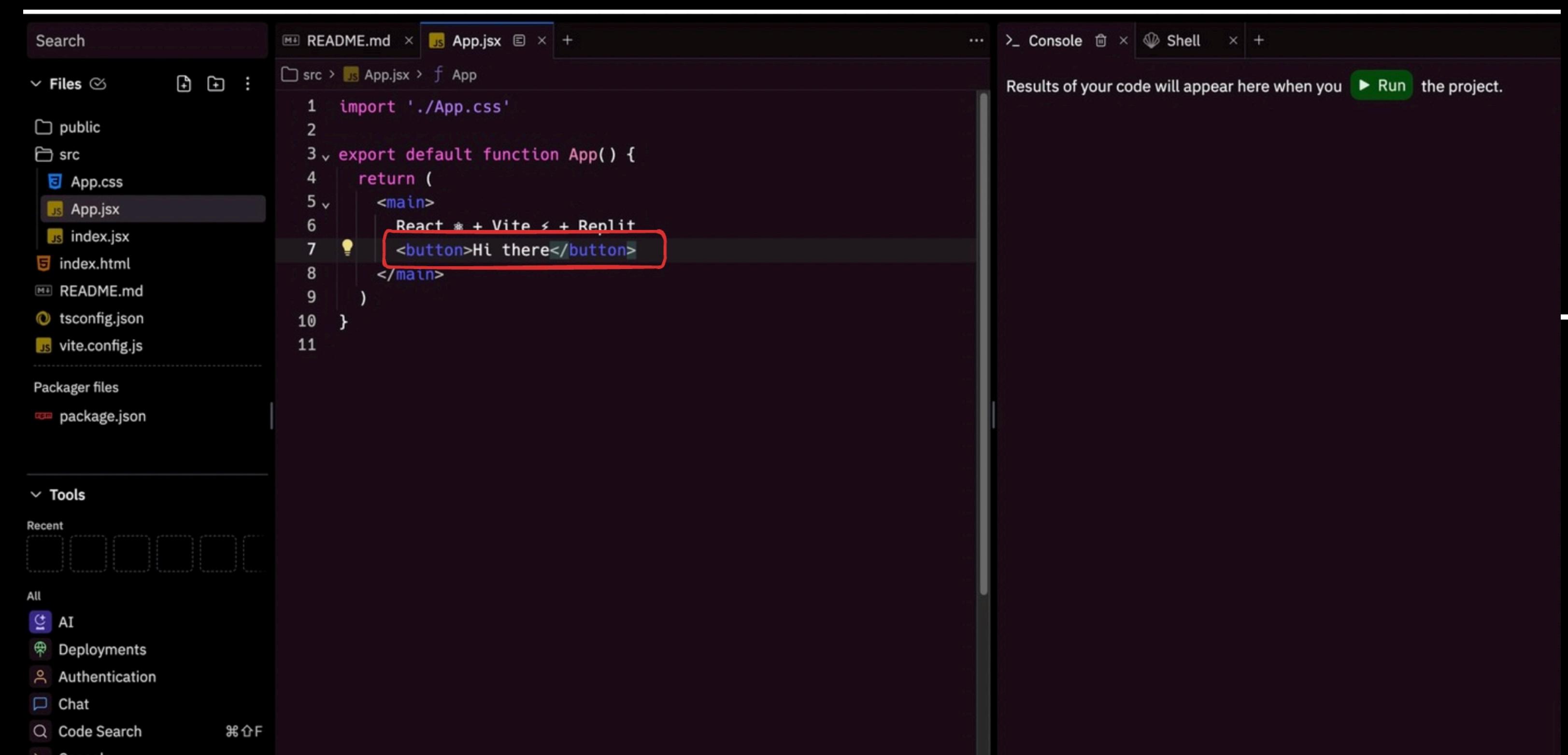
```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       React * + Vite < + Replit  
7       <button>Hi there</button>  
8     </main>  
9   )  
10 }  
11
```

The line containing the button element, `<button>Hi there</button>`, is highlighted with a red box. The code editor has a dark theme with light-colored syntax highlighting. On the right side of the editor, there is a preview area with the text "Results of your code will appear here when you Run the project." Below the editor, there is a "Tools" section with various icons and links.



Execution service

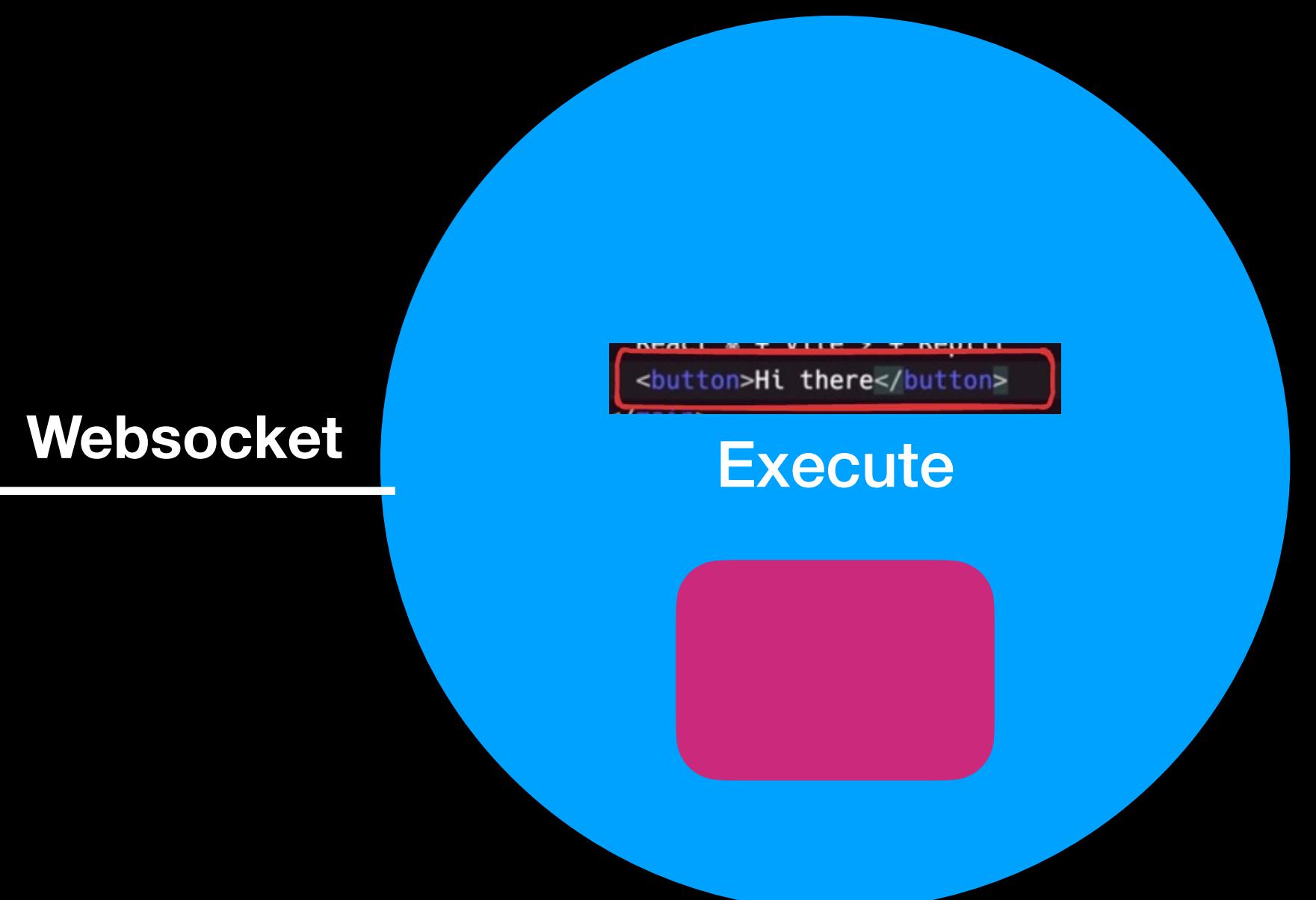
Step 5 - Let the user edit files



A screenshot of a code editor interface, likely Replit, showing a file named `App.jsx`. The code contains a simple React component:

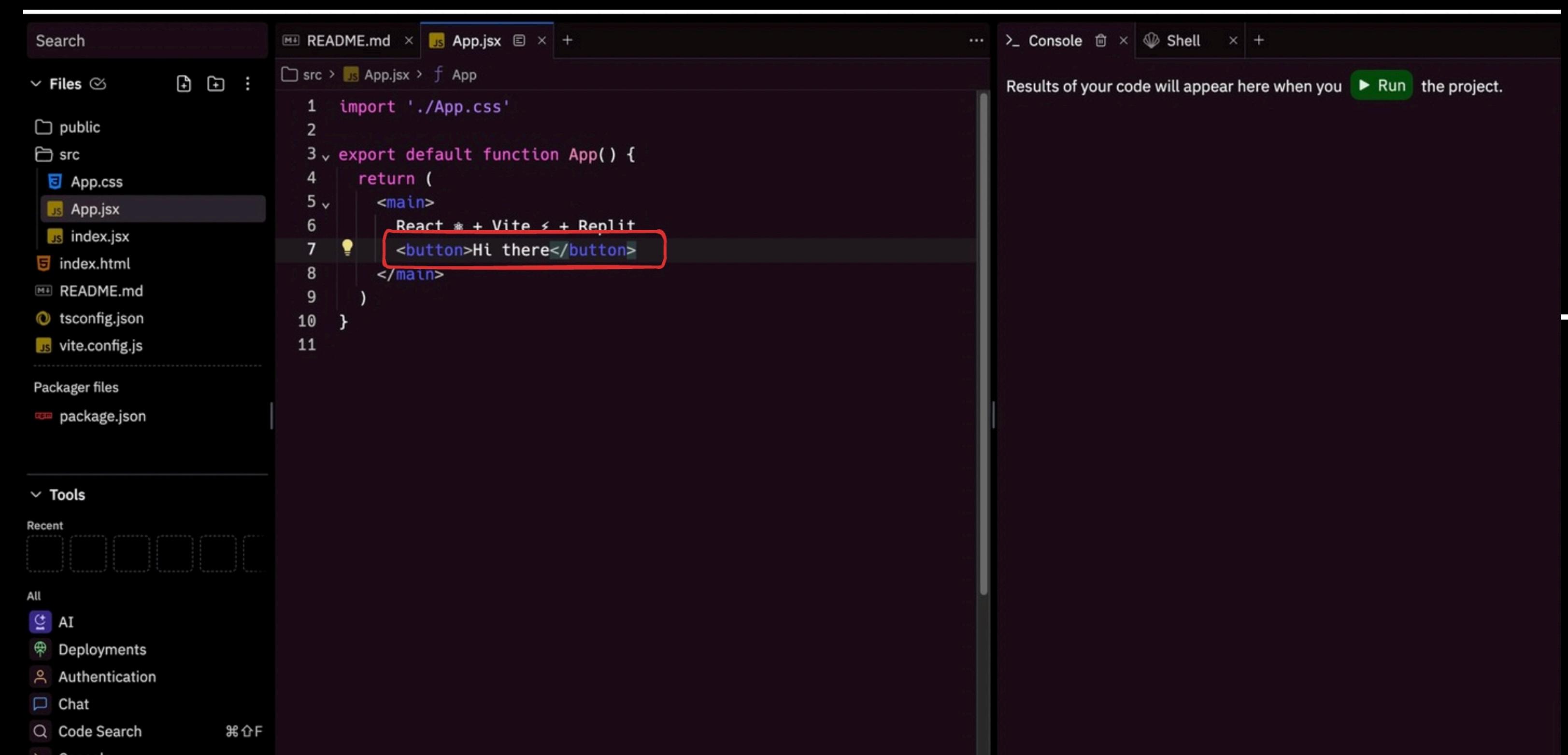
```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       React * + Vite * + Replit  
7       <button>Hi there</button>  
8     </main>  
9   )  
10 }  
11
```

The line containing the button element, `<button>Hi there</button>`, is highlighted with a red box. The code editor has a dark theme with light-colored syntax highlighting. A sidebar on the left shows project files like `public`, `src` (containing `App.css`, `App.jsx`, `index.js`, `index.html`), and `README.md`. A bottom bar includes links for AI, Deployments, Authentication, Chat, Code Search, and Console.



Execution service

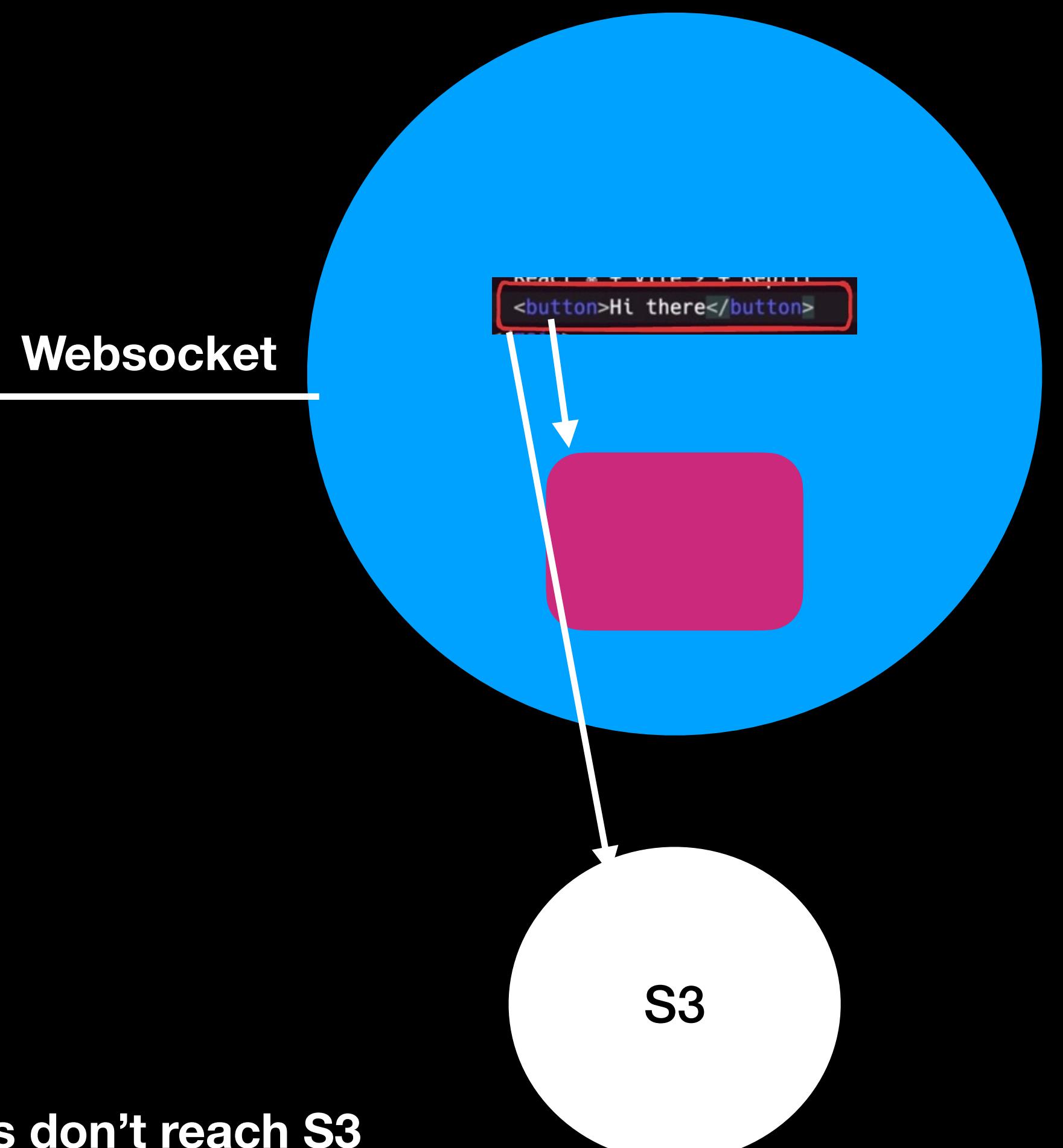
Step 5 - Let the user edit files



A screenshot of a code editor showing a file named App.jsx. The code contains a simple React component:

```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       React * + Vite * + Replit  
7       <button>Hi there</button>  
8     </main>  
9   )  
10 }  
11
```

The line containing the button element is highlighted with a red box.

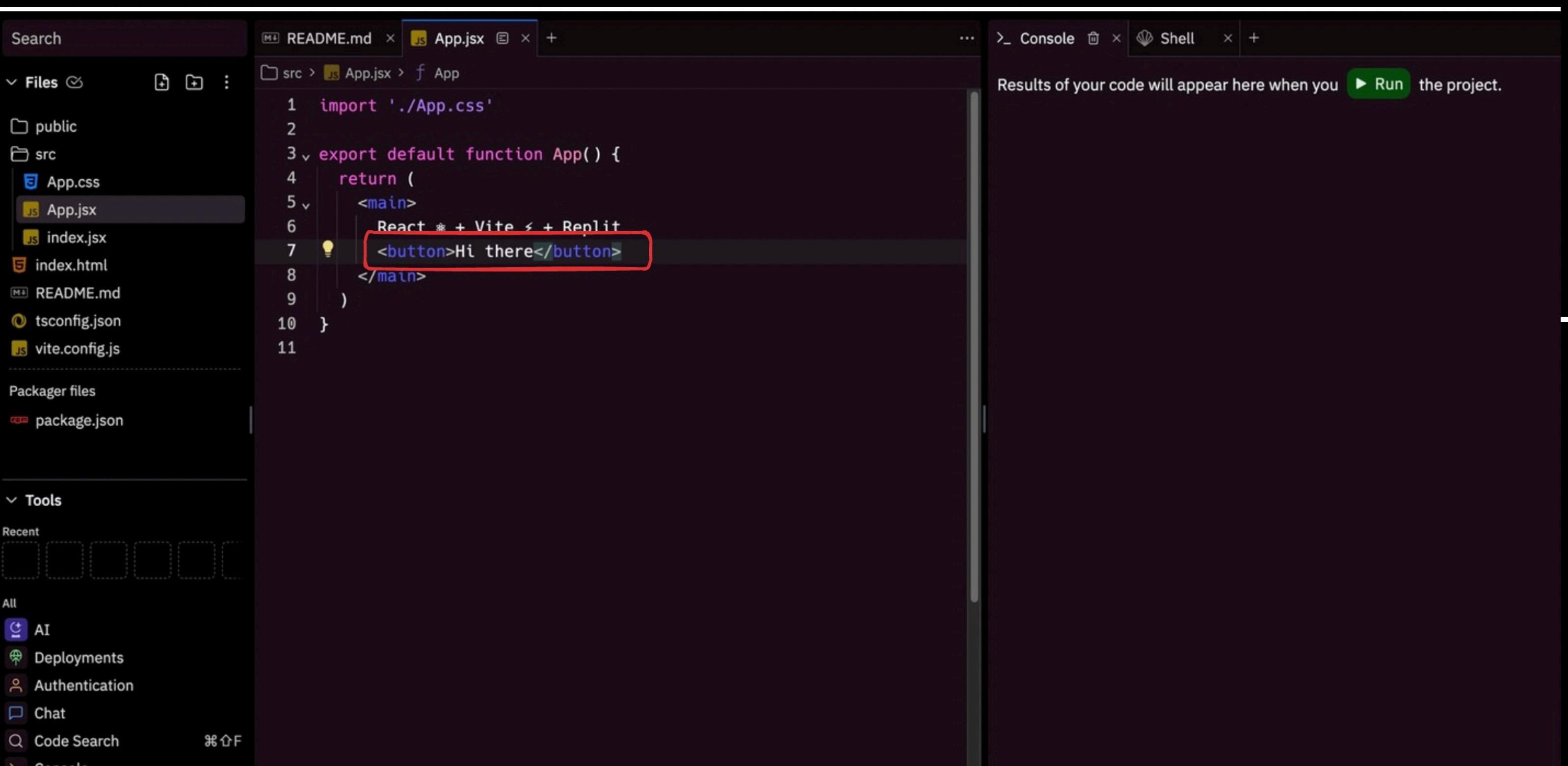


Callouts - Debounce these saves

You can mount a directory to S3 as well, although need to make sure node_modules don't reach S3

Execution service

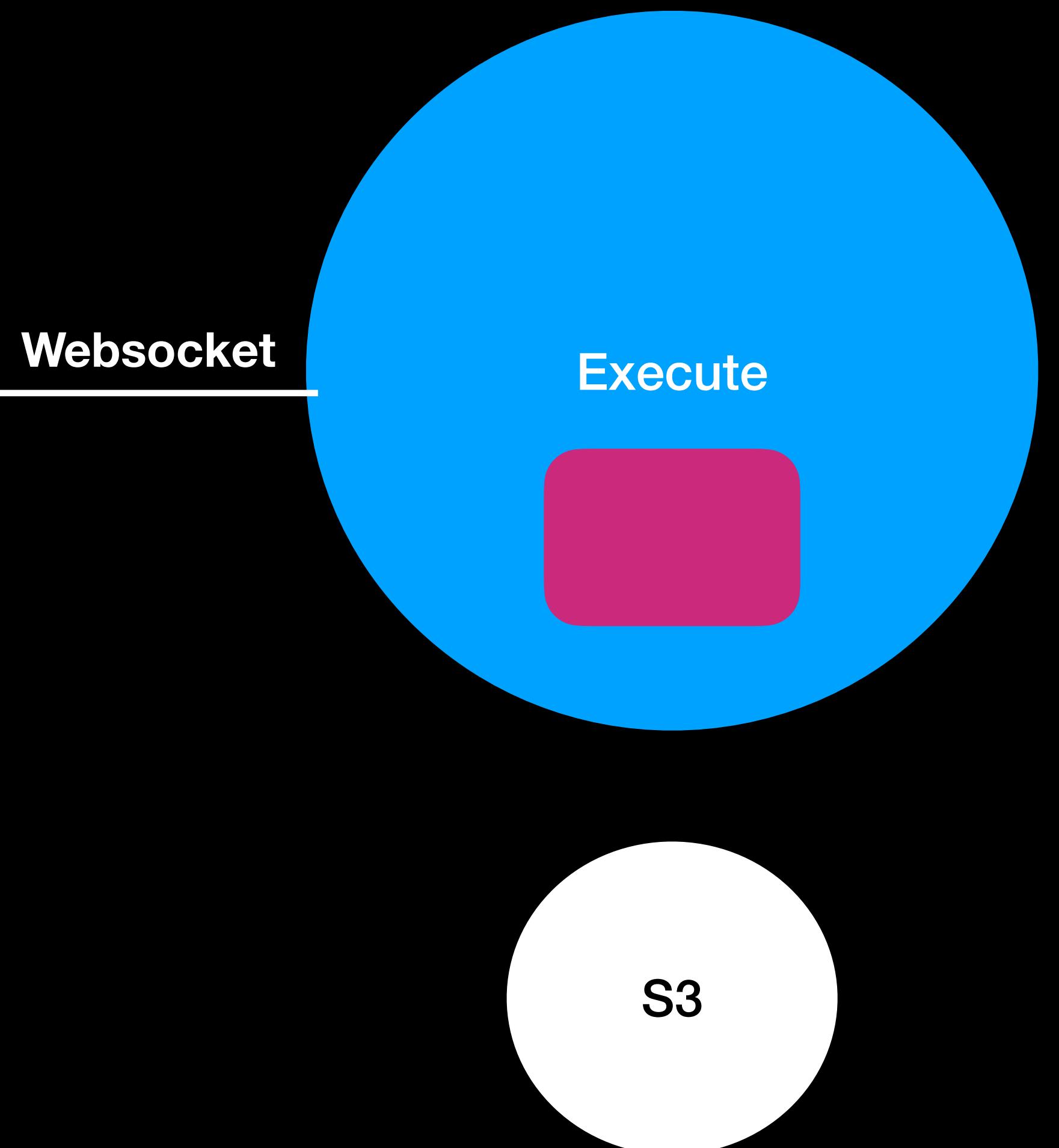
Logic to add and delete files also remains the same!
Validation of files (file format, size) is something you should take into consideration



A screenshot of a code editor interface, likely Replit, showing a file named App.jsx. The code contains a simple React component:

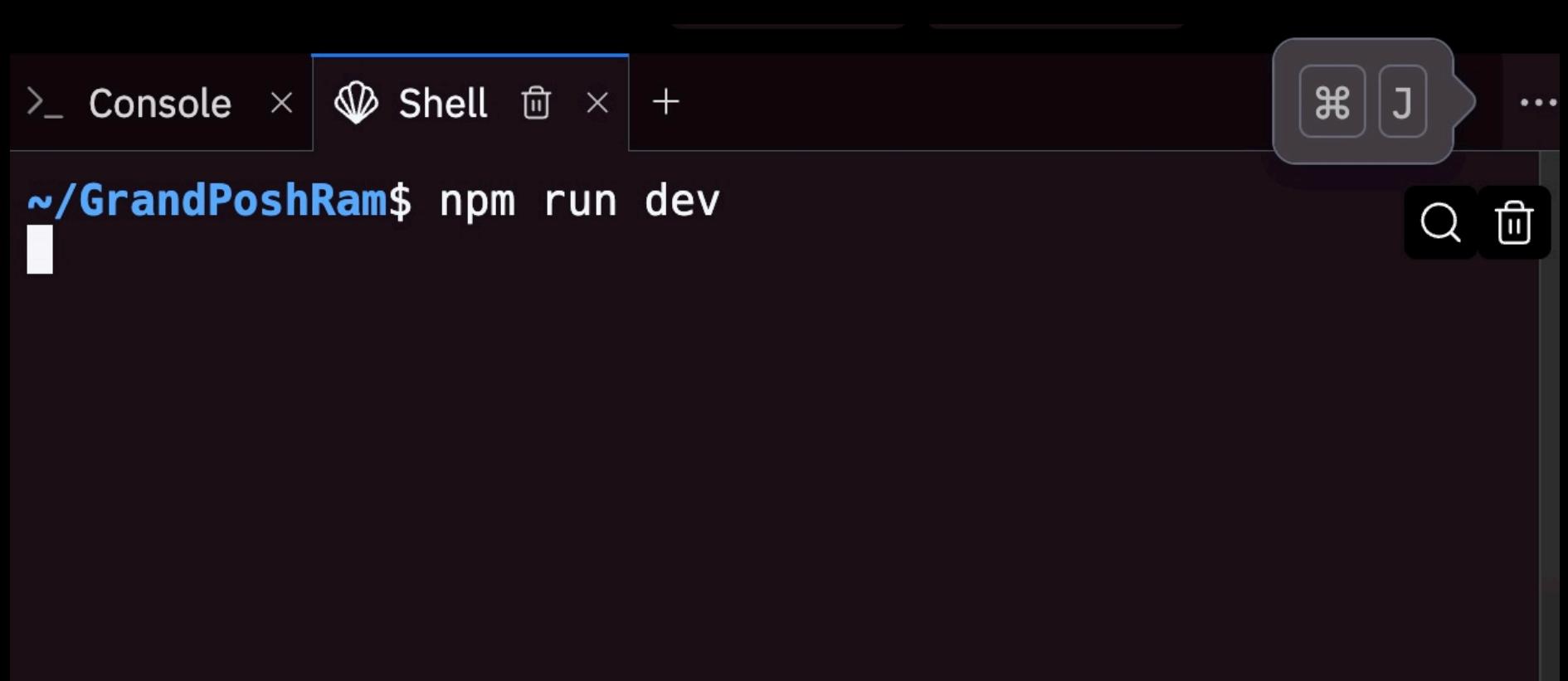
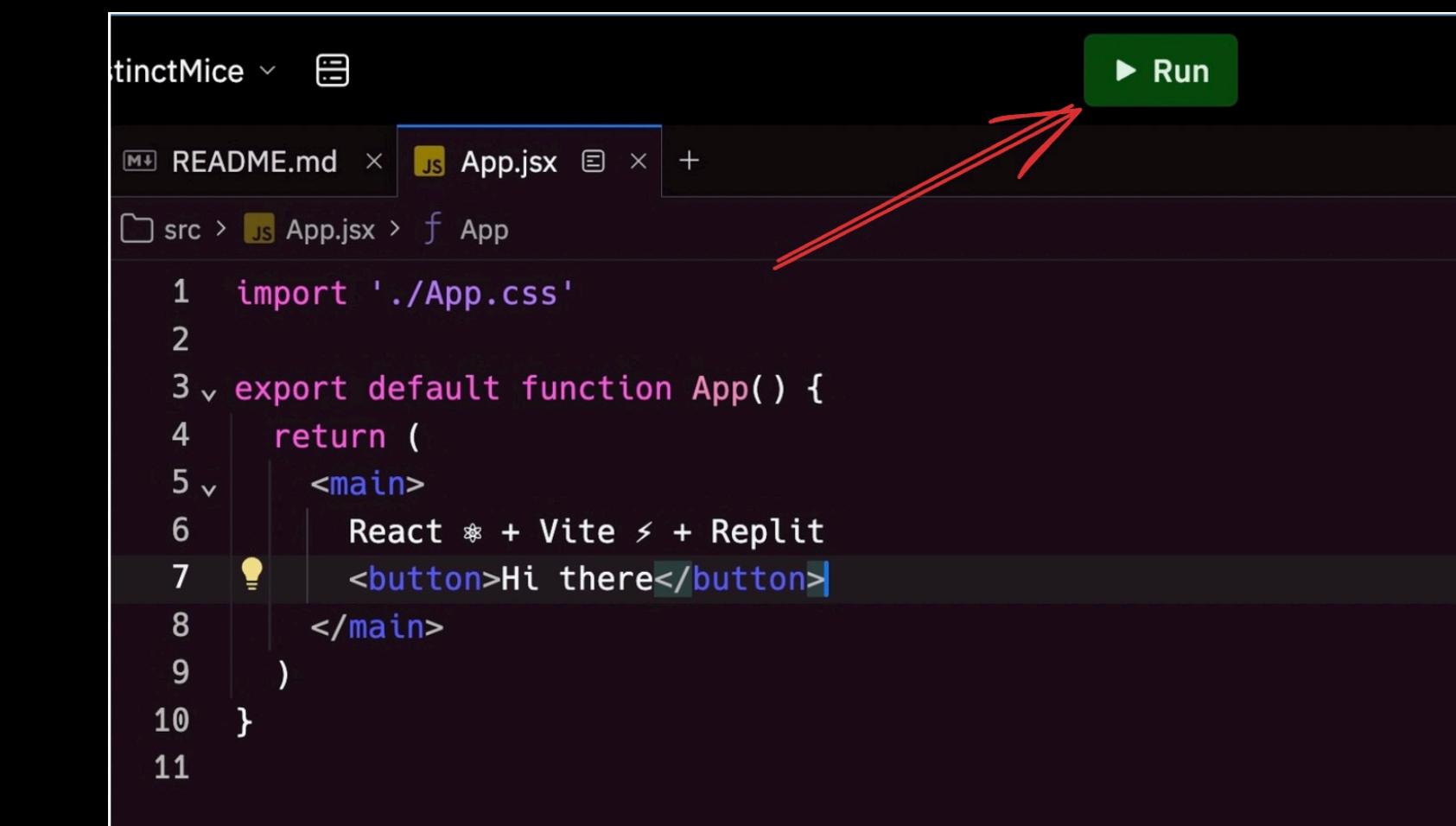
```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       React * + Vite * + Replit  
7       <button>Hi there</button>  
8     </main>  
9   )  
10 }  
11
```

The button element at line 7 is highlighted with a red box.



Execution service

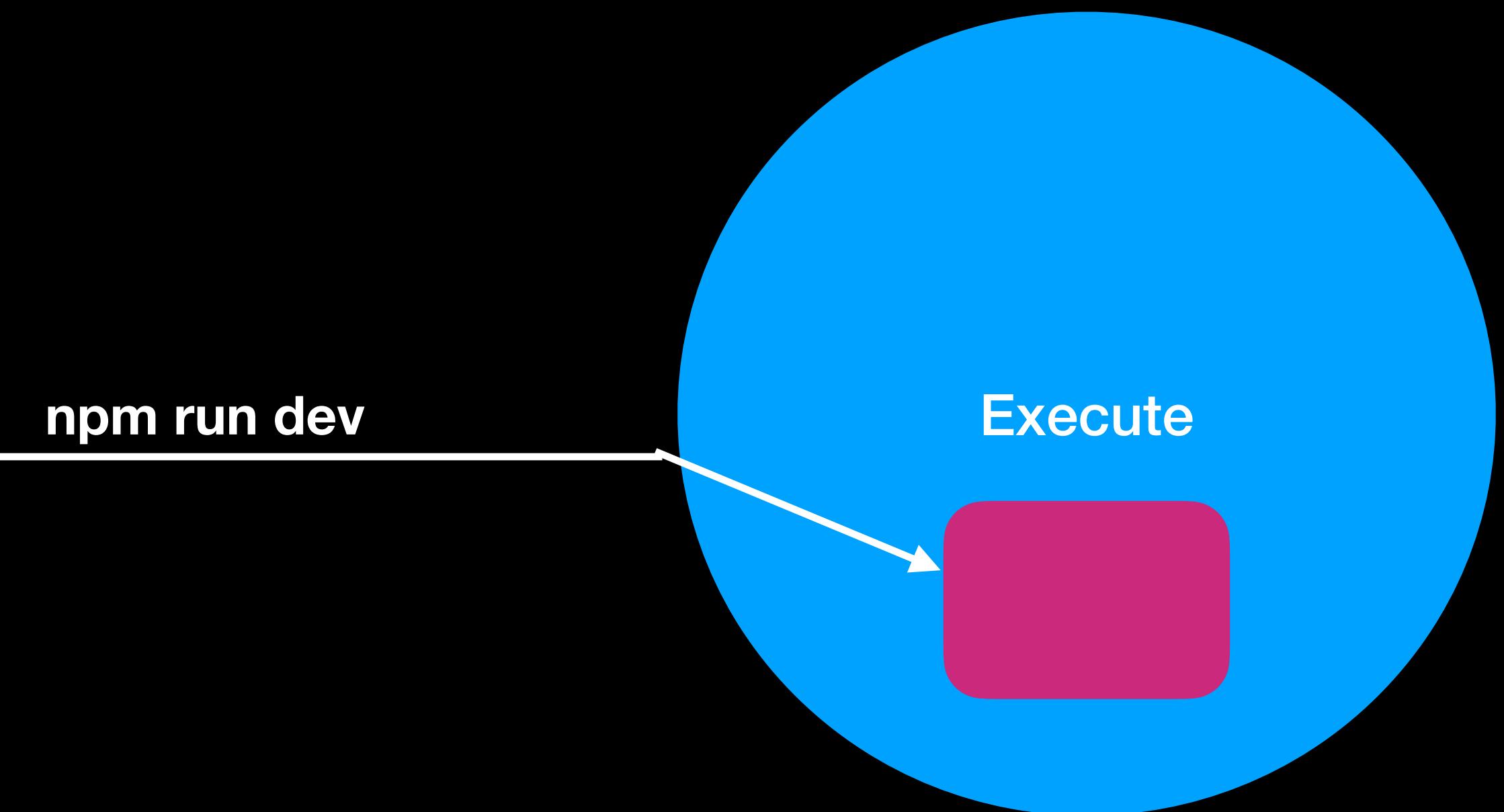
Step 6 -Running/Executing the code



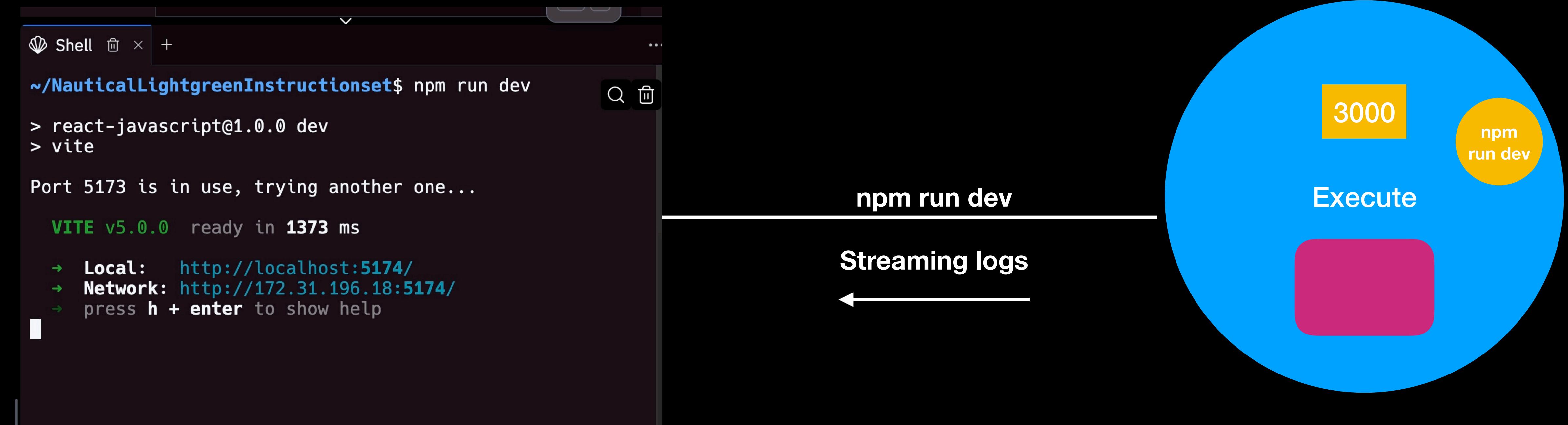
A screenshot of a code editor (VS Code) showing the file `App.jsx`. The code contains a simple React component definition:1 import './App.css'
2
3 export default function App() {
4 return (
5 <main>
6 React * + Vite < + Replit
7 <button>Hi there</button>
8 </main>
9)
10 }
11A red arrow points from the top of the slide towards the green `Run` button in the top right corner of the code editor interface.

Console output:

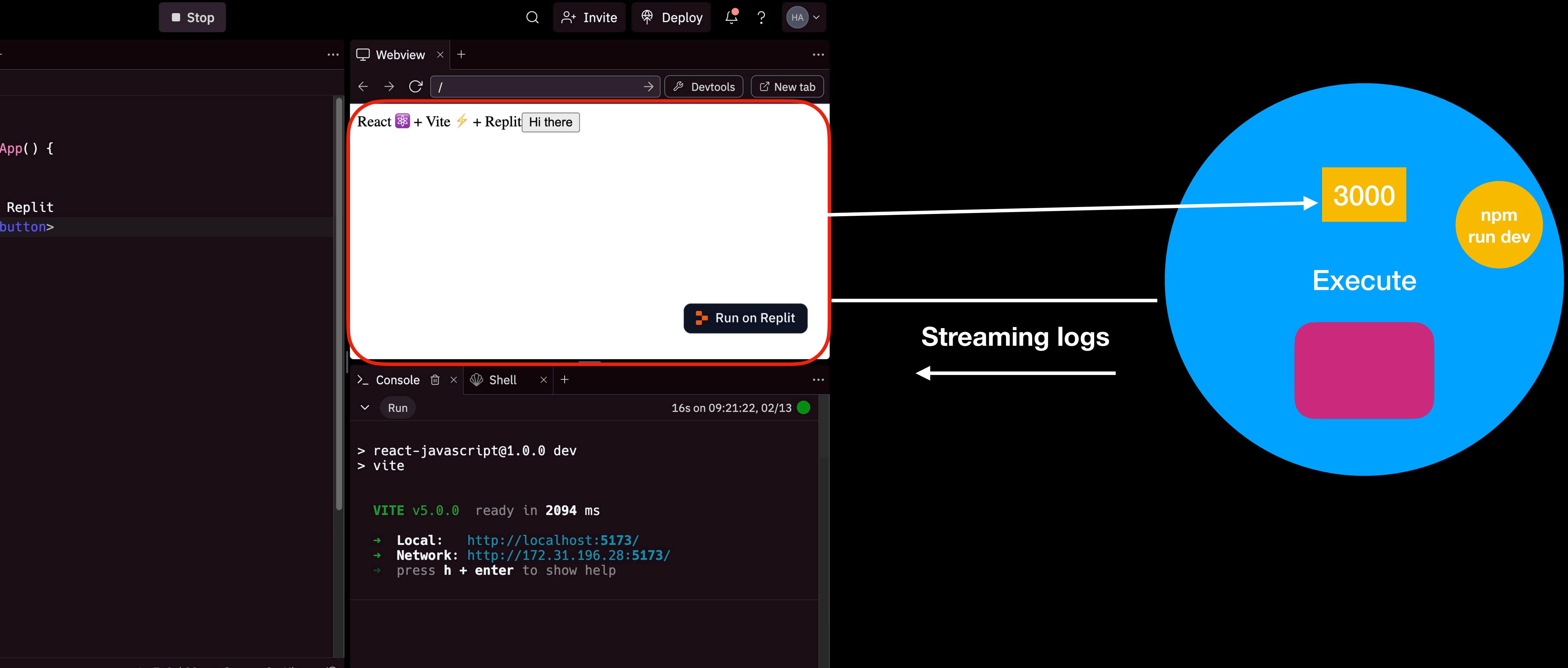
```
~/GrandPoshRam$ npm run dev
```



Execution service



Execution service



Execution service

A screenshot of a development environment showing a code editor, a browser preview, and a terminal window.

- Code Editor:** Shows a file named `App.jsx` with the following content:

```
1 import './App.css'
2
3 export default function App() {
4   return (
5     <main>
6       React * + Vite * + Replit
7       <button>Hi there</button>
8     </main>
9   )
10 }
```
- Browser Preview:** Shows a simple React application with the text "Hi there".
- Terminal:** Shows the command `vite` being run, resulting in a Vite instance ready in 2094 ms, with local and network URLs provided.

Disconnects

Execute

Clean up resources

1. Wait for a bit before removing the folder
2. Flush to S3
3. Stop any lingering process

You can wait for a bit before doing this

Downsides

- 1. Remote code execution**
- 2. Single server setup, doesn't autoscale**
- 3. Port conflicts between two users (every user is sharing resources on the same server)**
- 4. Terminal is very ad-hoc/first principles**

Code

Disclaimers

1. I'm using Node.js. Keep it simple
2. I'm using socket.io. Keep it simple
3. I'll be writing code in TS, but nothing too strict. Keep it simple
4. I will not be adding any extra fluff that's not needed for this tutorial (eslint, prettier).
5. No monorepos - code repetition

Should you create a zig based well linted 100% tested CI/CD implemented system?

Maybe

But we're limiting the scope of this tutorial

Introducing PTY

If you want to do a terminal inside a browser
(or create your own terminal for eg)

What you can do is create a `pseudo terminal`
that your browser can talk to

xterm.js is a library that lets you fetch and
forward keystrokes to a `pseudo terminal` that
you can spawn on your server

The screenshot shows the official Xterm.js website. At the top is the Xterm.js logo, which consists of a stylized 'X' icon followed by the text 'term.js'. Below the logo is the tagline 'Build terminals in the browser'. A button labeled 'npm install xterm' with a 'Copy' link is present. A dark callout box contains the text: 'Xterm.js is the frontend component that powers many terminals including VS Code, Hyper and Theia!'. It lists several features: 'Apps just work' (Xterm.js works with most terminal apps like bash, vim and tmux), 'Accessible' (A screen reader mode is available), 'Unicode support' (Supports CJK 語 and emoji ♥), 'Performance' (Xterm.js is fast and includes an optional WebGL renderer), and 'Self-contained' (Zero external dependencies). It also mentions 'And much more...' with links to 'Links', 'themes', 'addons', 'typed API', and 'decorations'. A note at the bottom says 'Try clicking italic text' with a small button. Below the callout box is a terminal window showing the text: 'Below is a simple emulated backend, try running `help`.' followed by a '\$' prompt.

<https://github.com/xtermjs/xterm.js>

<https://github.com/replit/ruspty>

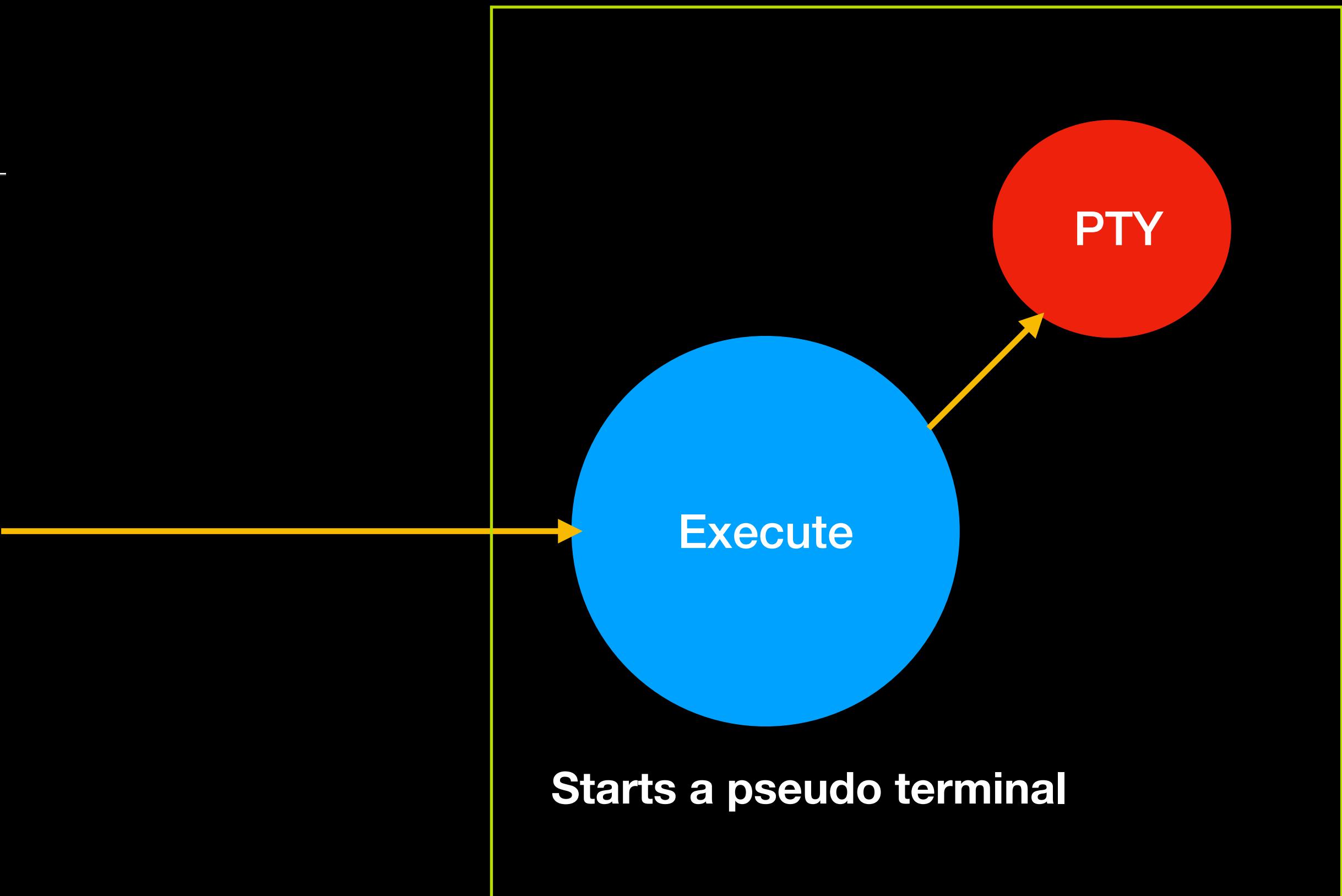
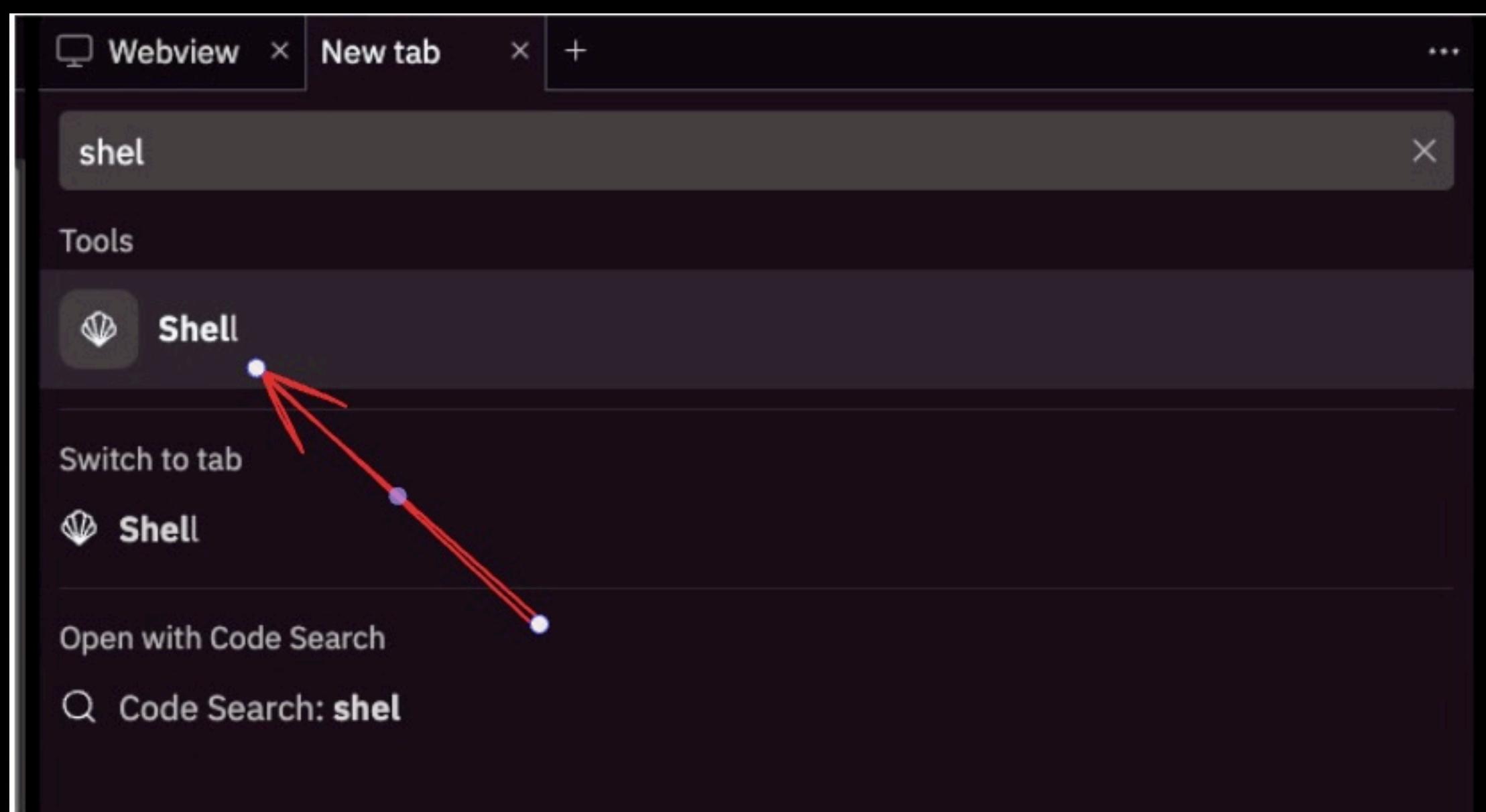
Introducing PTY

Old approach



Introducing PTY

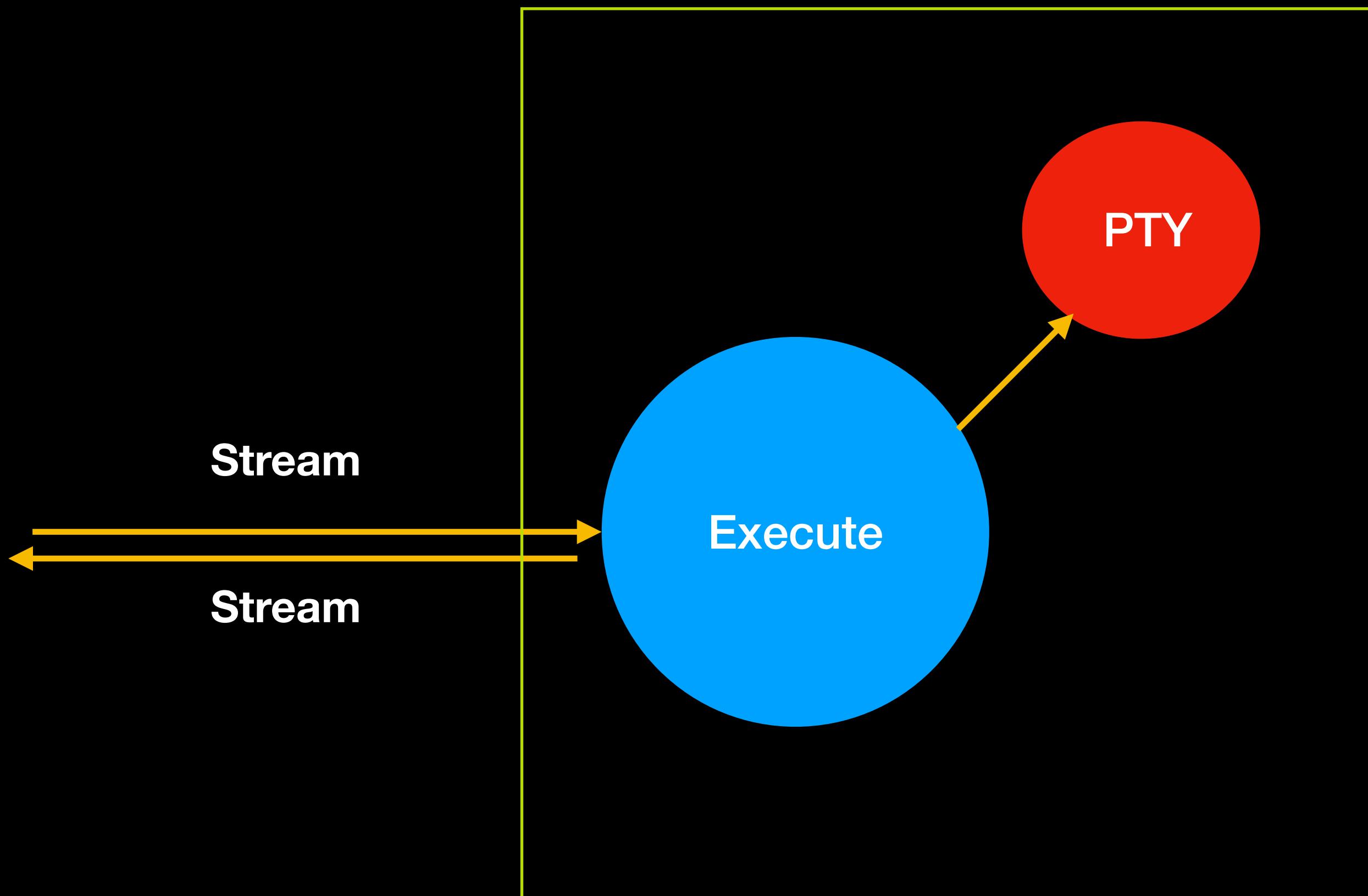
New approach



Introducing PTY

New approach

```
# llss
app      cron          Gemfile      log        spec
bin      db            Gemfile.lock  public     tmp
config   docker-entrypoint.sh lib        Rakefile   vendor
config.ru docs          LICENSE      scripts
#
app      cron          Gemfile      log        spec
bin      db            Gemfile.lock  public     tmp
config   docker-entrypoint.sh lib        Rakefile   vendor
config.ru docs          LICENSE      scripts
# ■
```



Part 2 | The good solution

Part 2 | The good solution

What were the biggest problems in approach #1 ?

- 1. Remote code execution isn't safe**
- 2. Doesn't autoscale**

Part 2 | The good solution

There are two approaches you can take

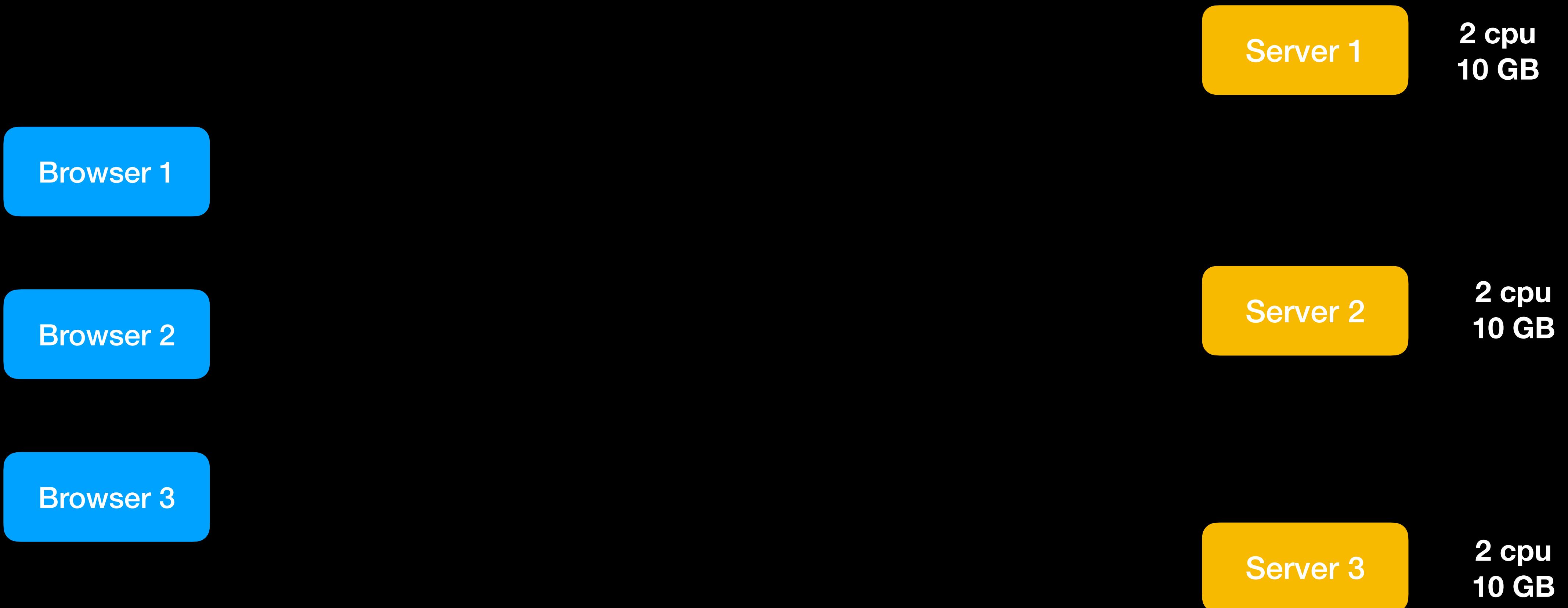
Cloud specific Autoscaling

AWS - Auto scaling Groups, ECS

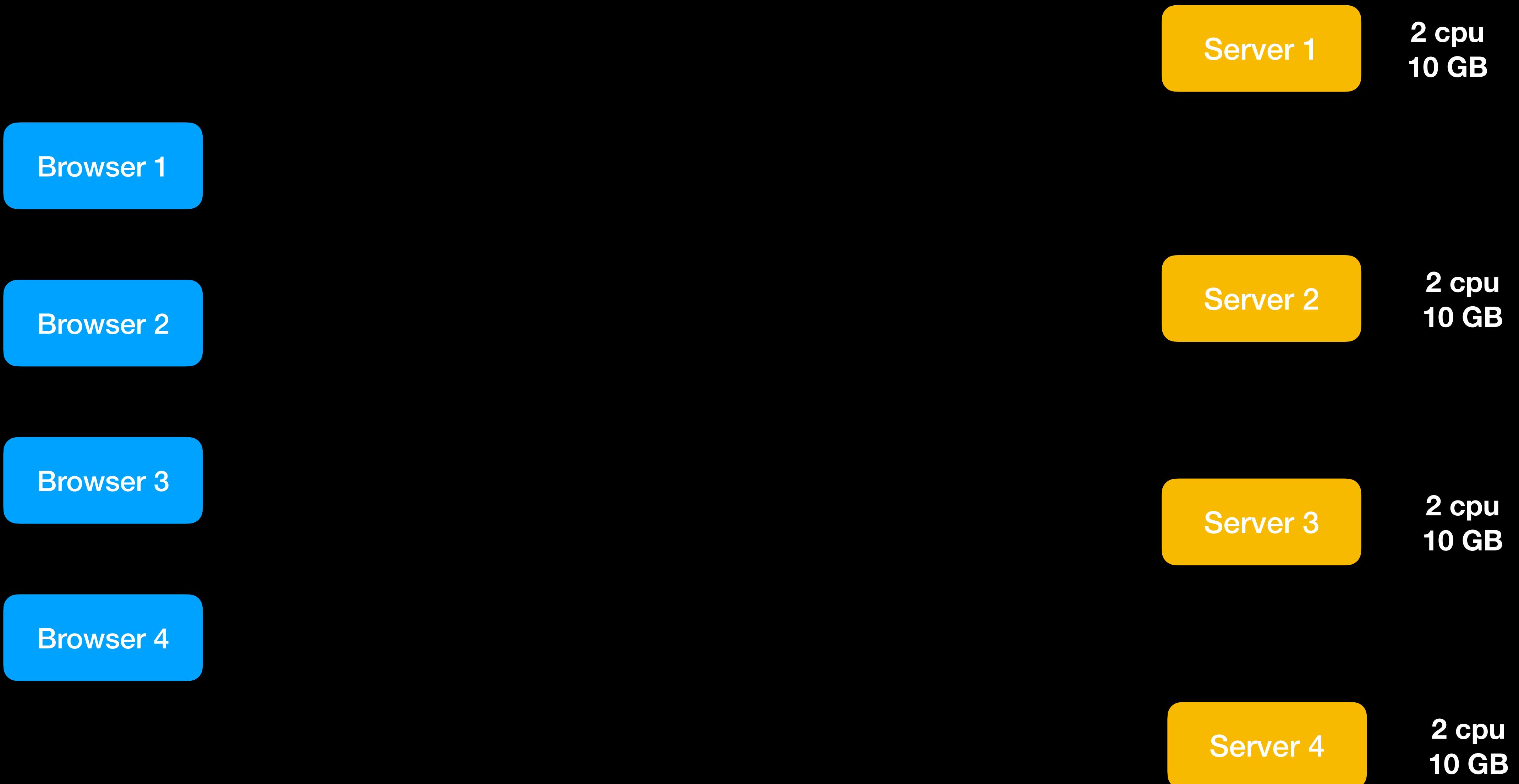
Container Orchestration

Kubernetes

Cloud specific Autoscaling



Cloud specific Autoscaling



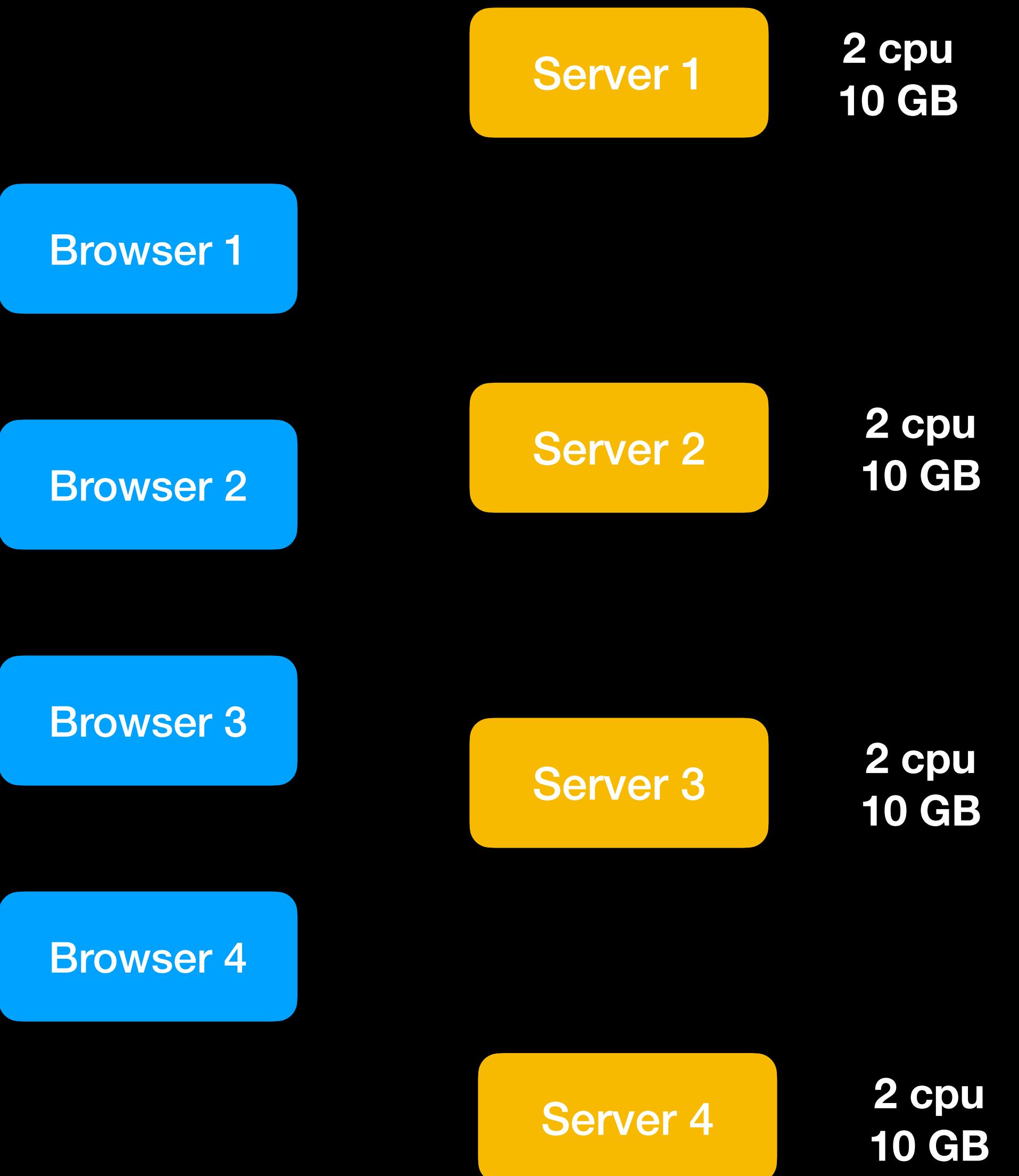
Cloud specific Autoscaling

Upsides

1. Easy to do
2. Provides you a way to securely run code
3. Autoscales
4. No port conflicts

Downsides

1. Bootup time (not a huge problem)
2. Over provisioned servers
3. Not cloud agnostic



Kubernetes

Before we get into why this approach is good, let's understand what

- 1. Containers are**
- 2. k8s is**

Containers

Containers

Mac

```
~ (-zsh)
~ node index.js

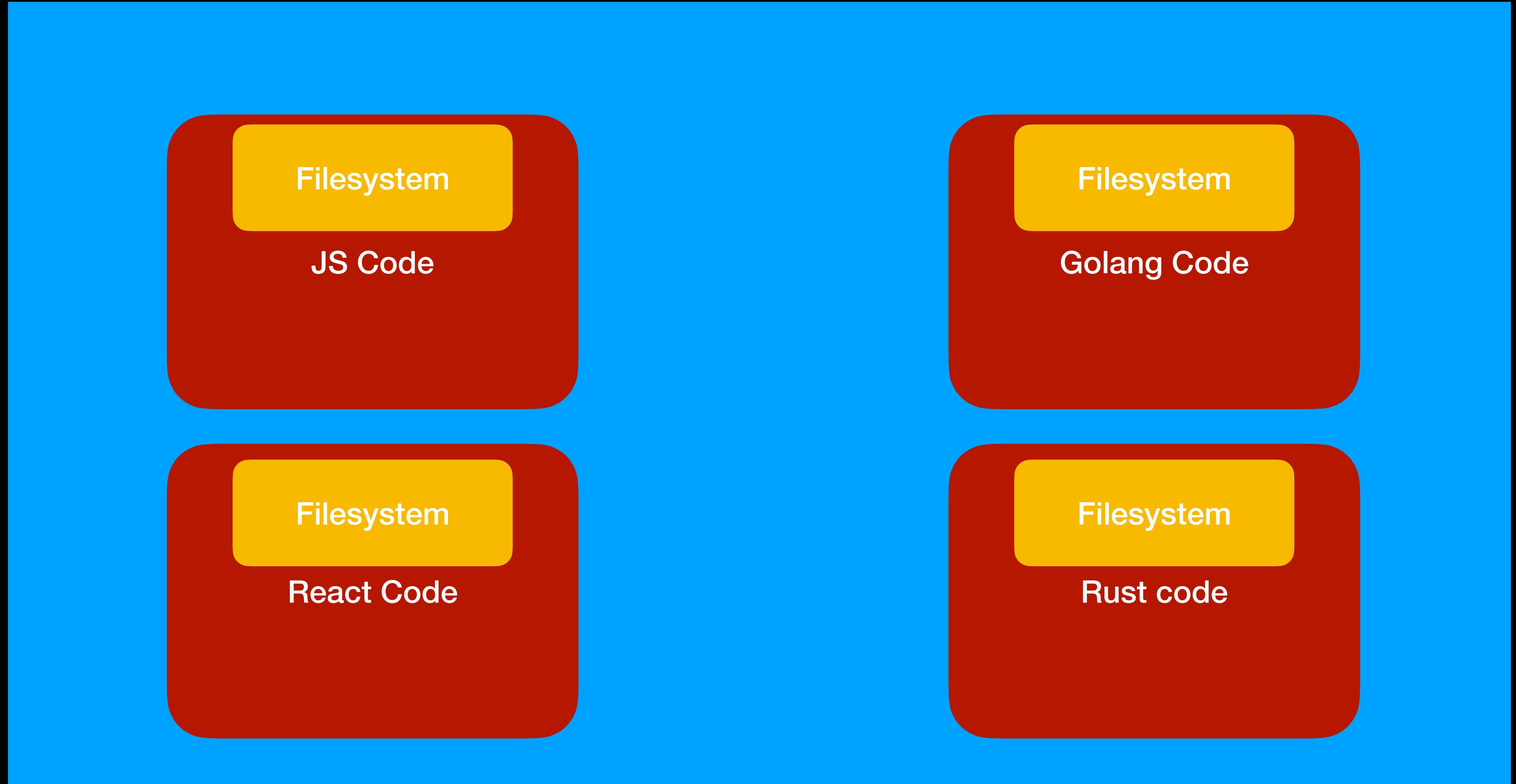
~ (-zsh)
~ go run main.go

~ (-zsh)
Last login: Wed Feb 14 08:59:04 on ttys005
~ npm run dev

~ (-zsh)
~ cargo run
```

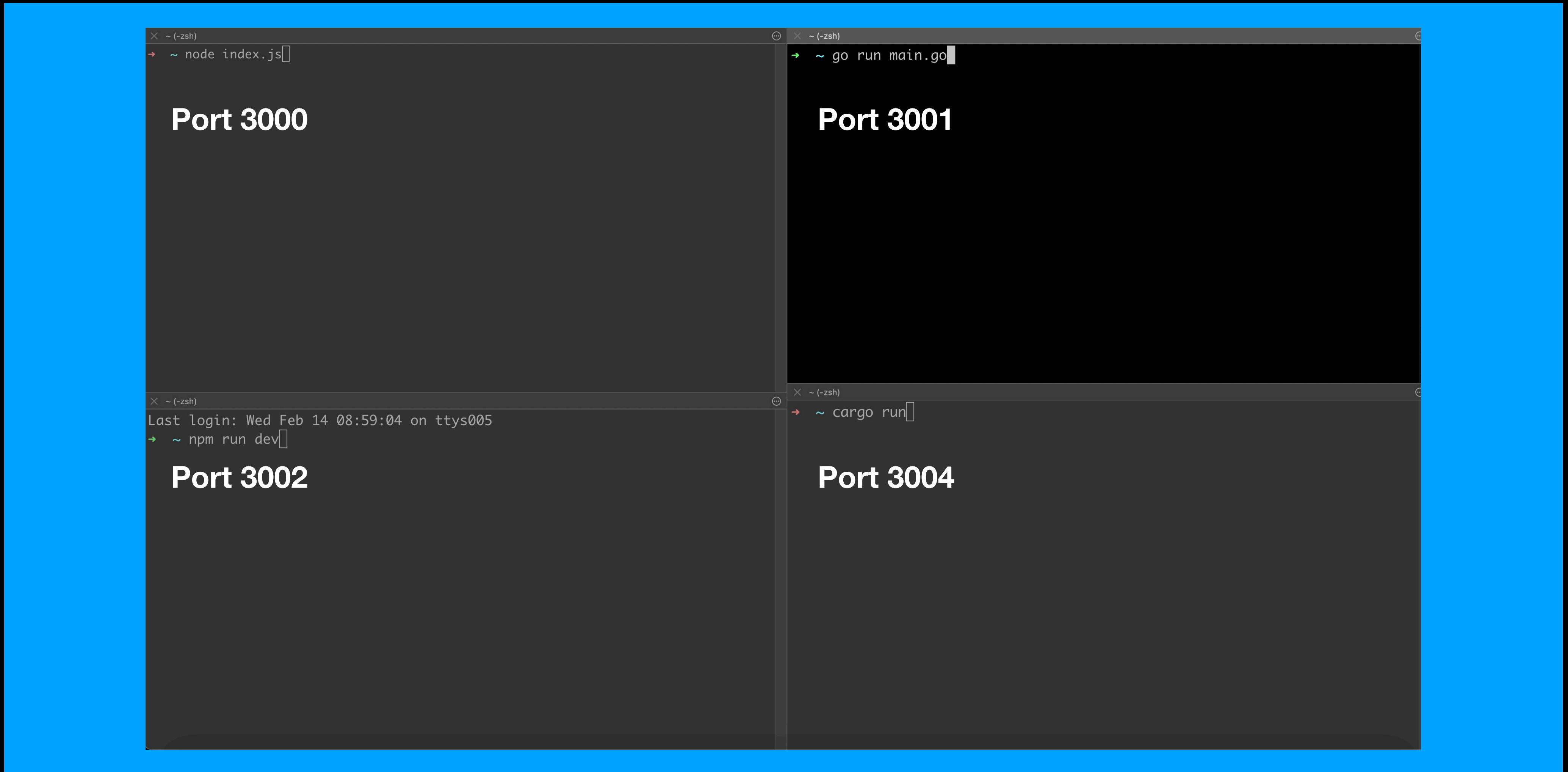
Containers

Mac



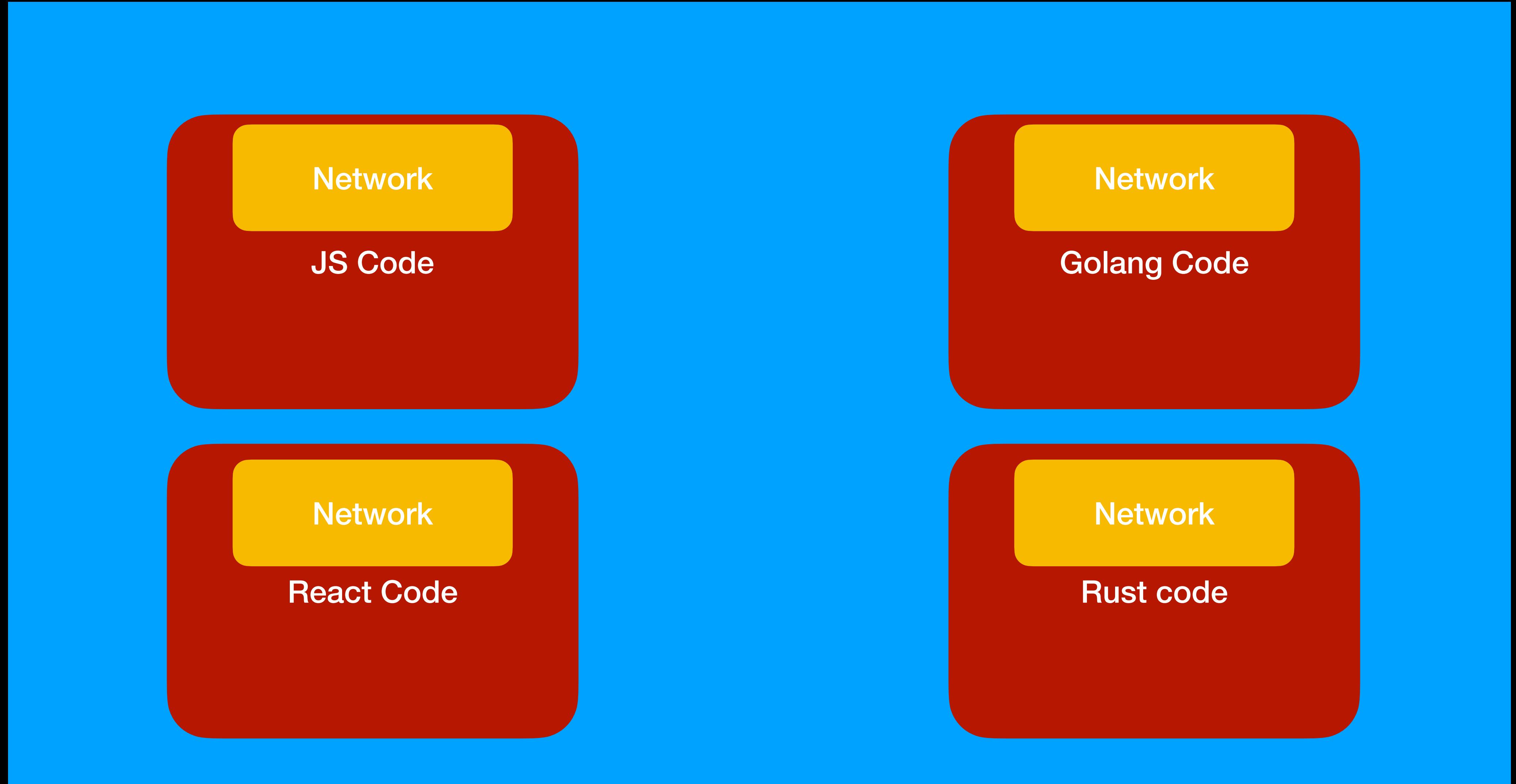
Containers

Mac



Containers

Mac



Container Orchestration - Kubernetes

**What if you want to run multiple such containers,
and autoscale your servers?**

Container Orchestration - Kubernetes

Mac

Linux

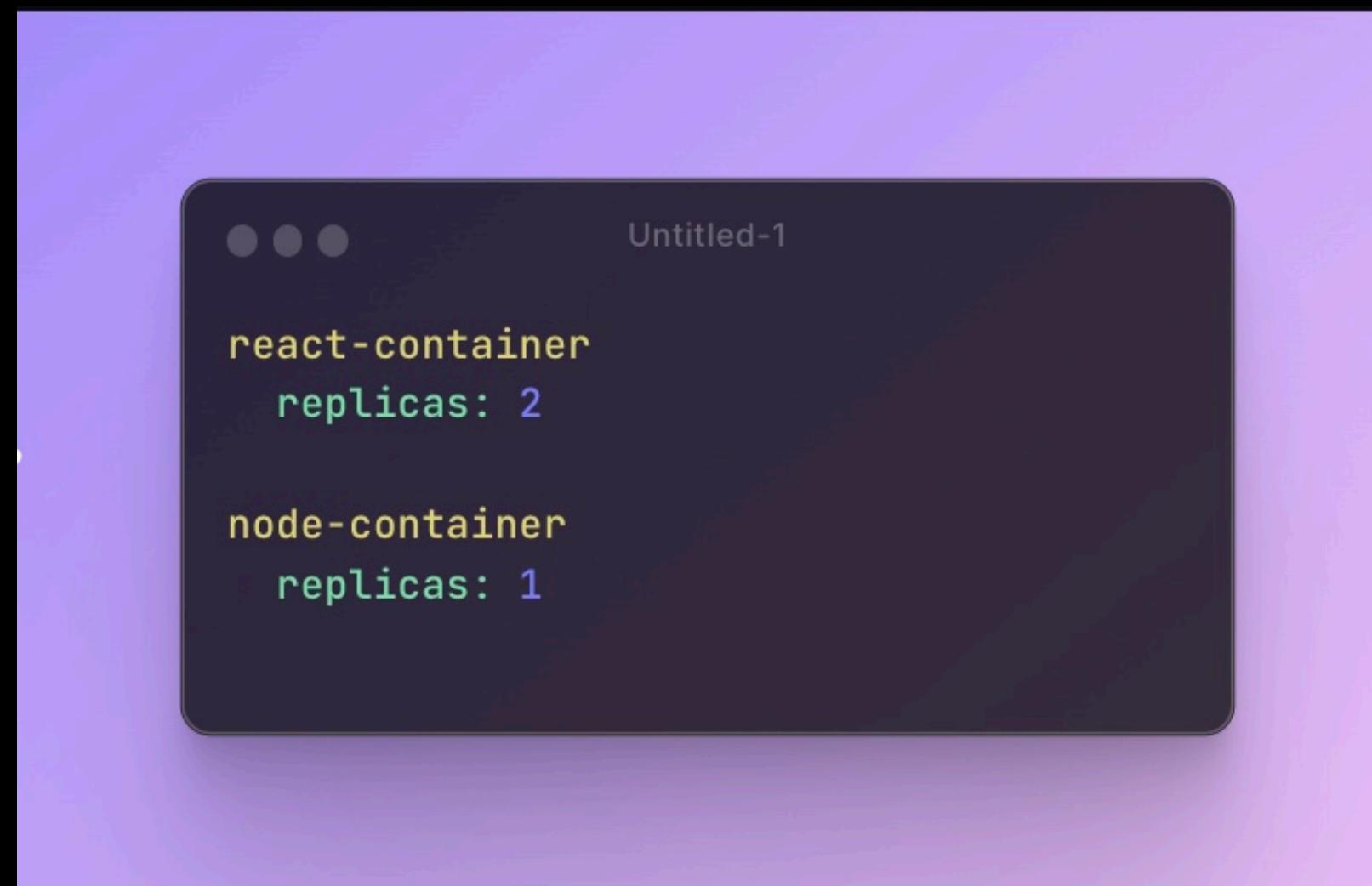
**What if you could have a cluster of machines
and describe in a single file how to
run multiple containers**

Windows

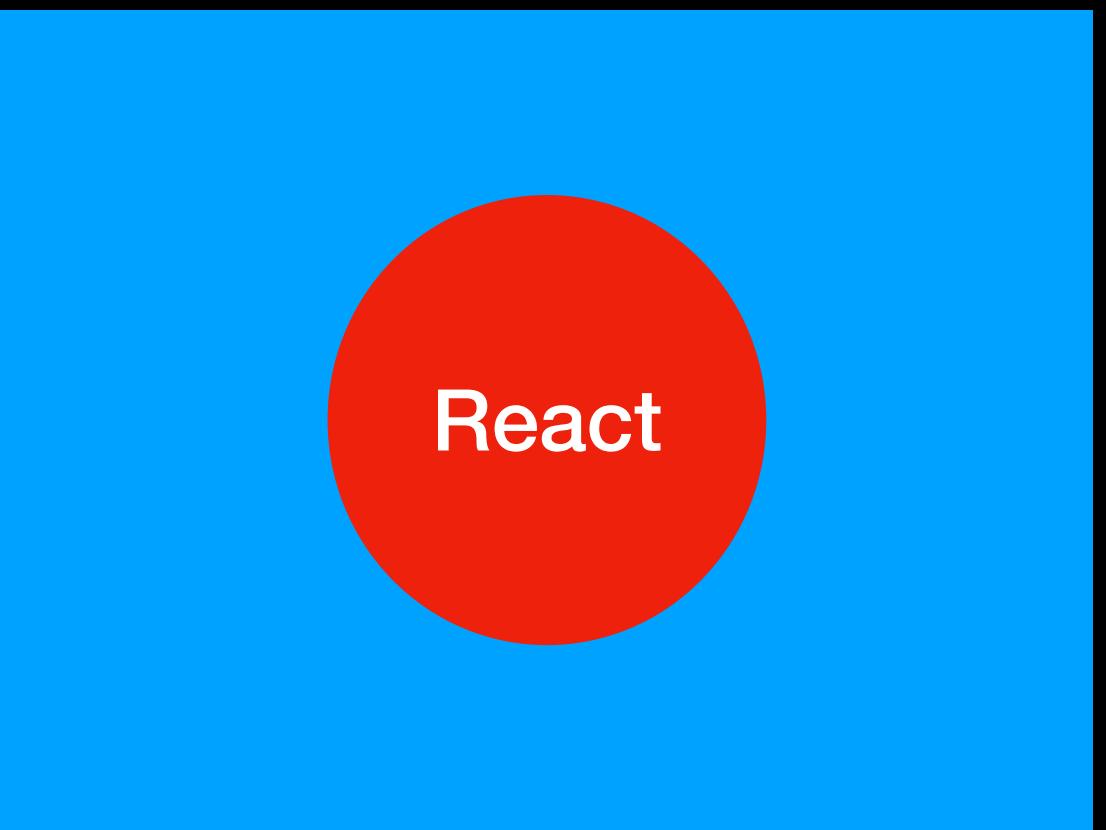
Ubuntu

Container Orchestration - Kubernetes

What if you could have a cluster of machines
and describe in a single file how to
run multiple containers



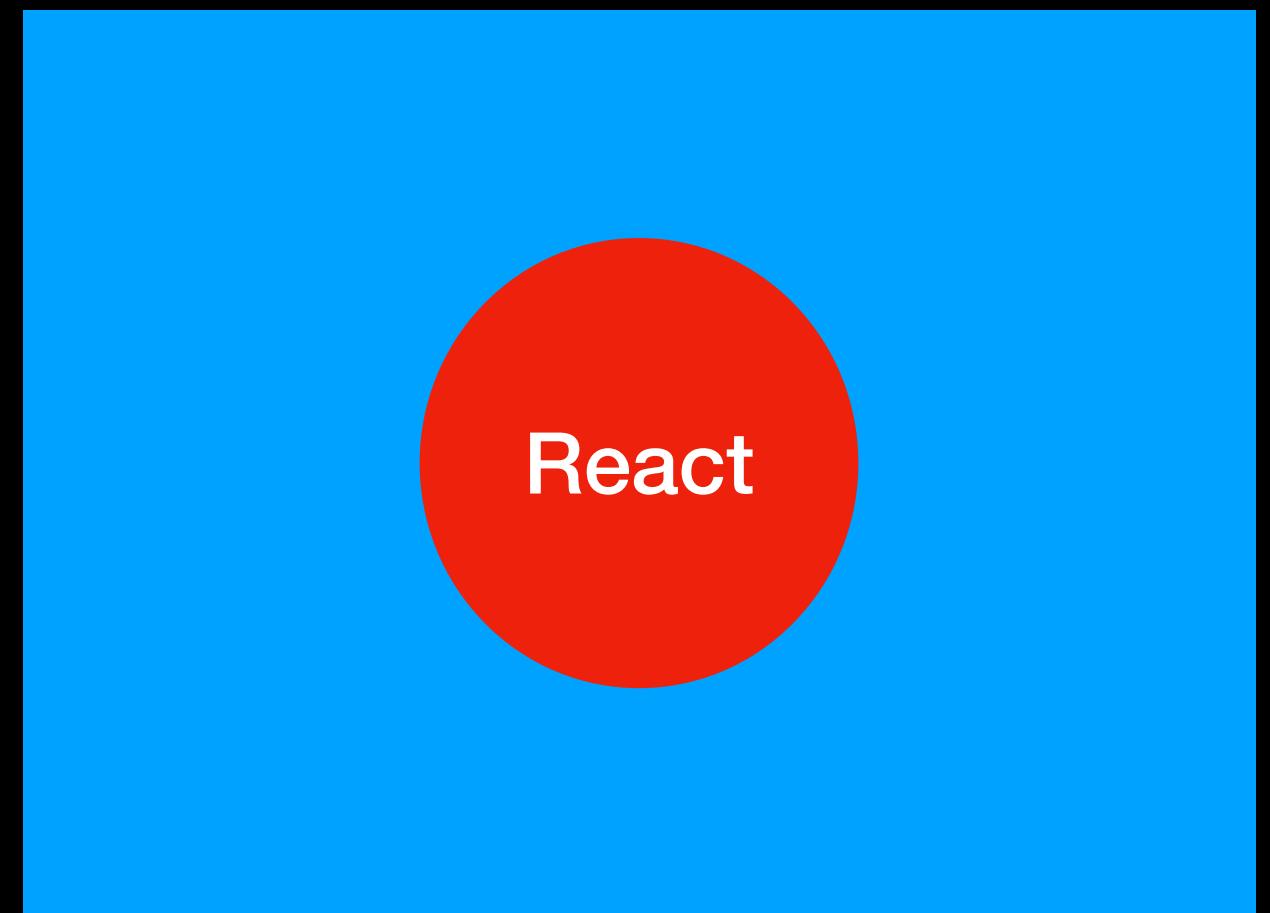
Mac



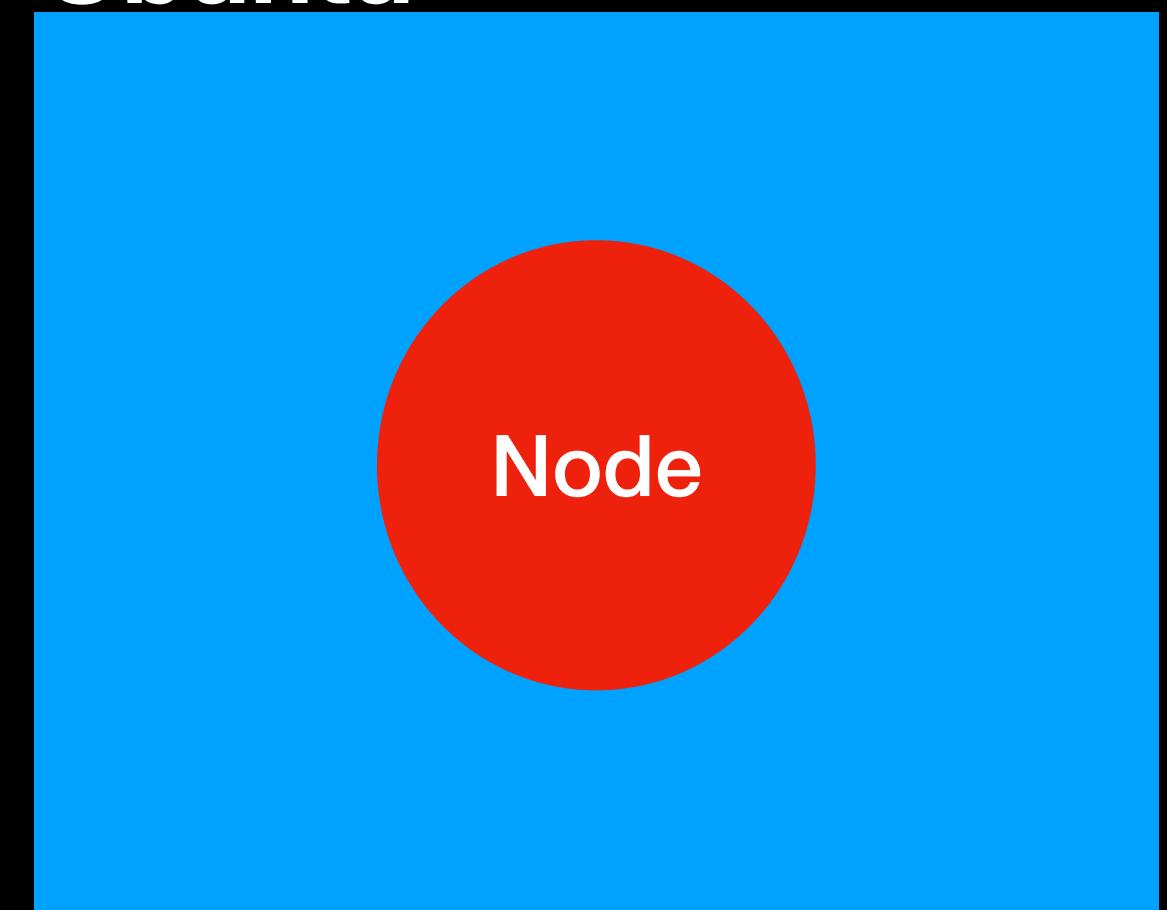
Windows



Linux

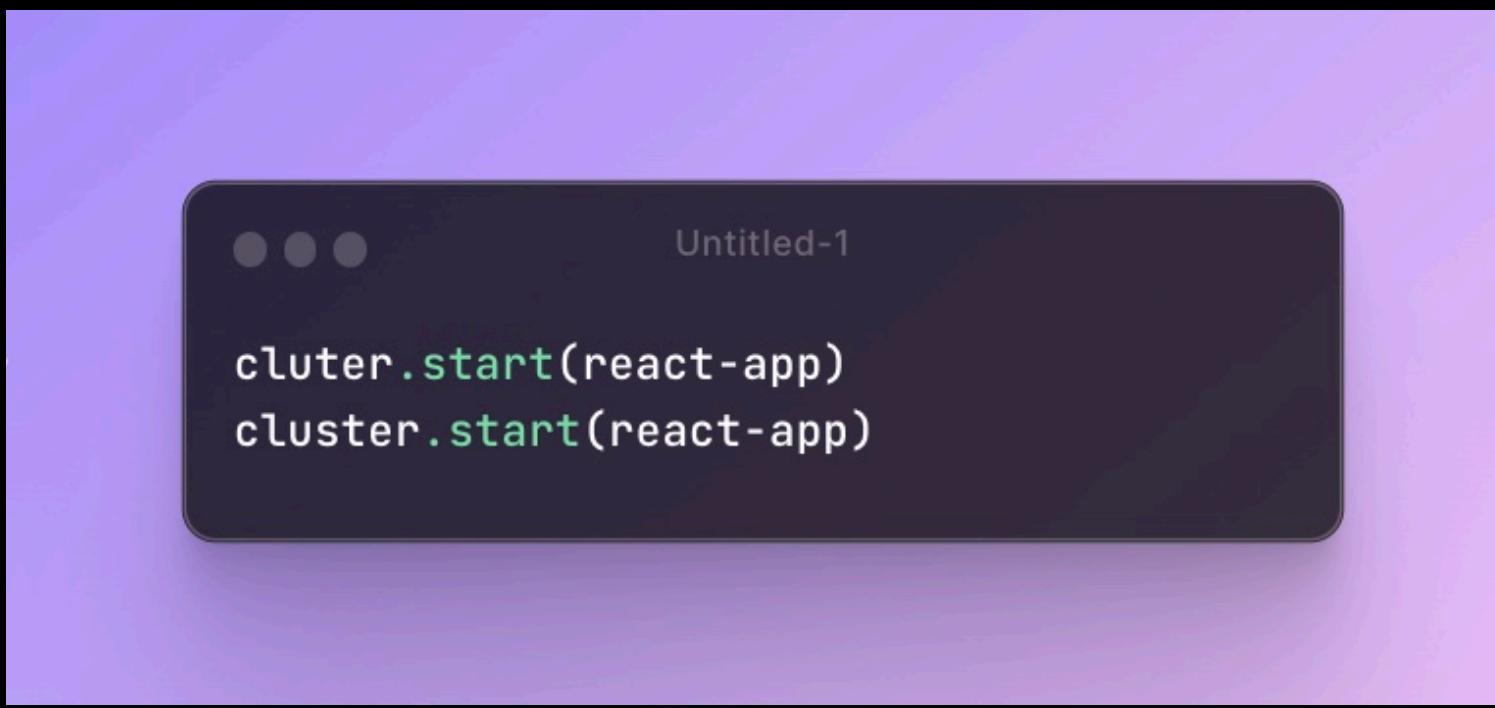


Ubuntu

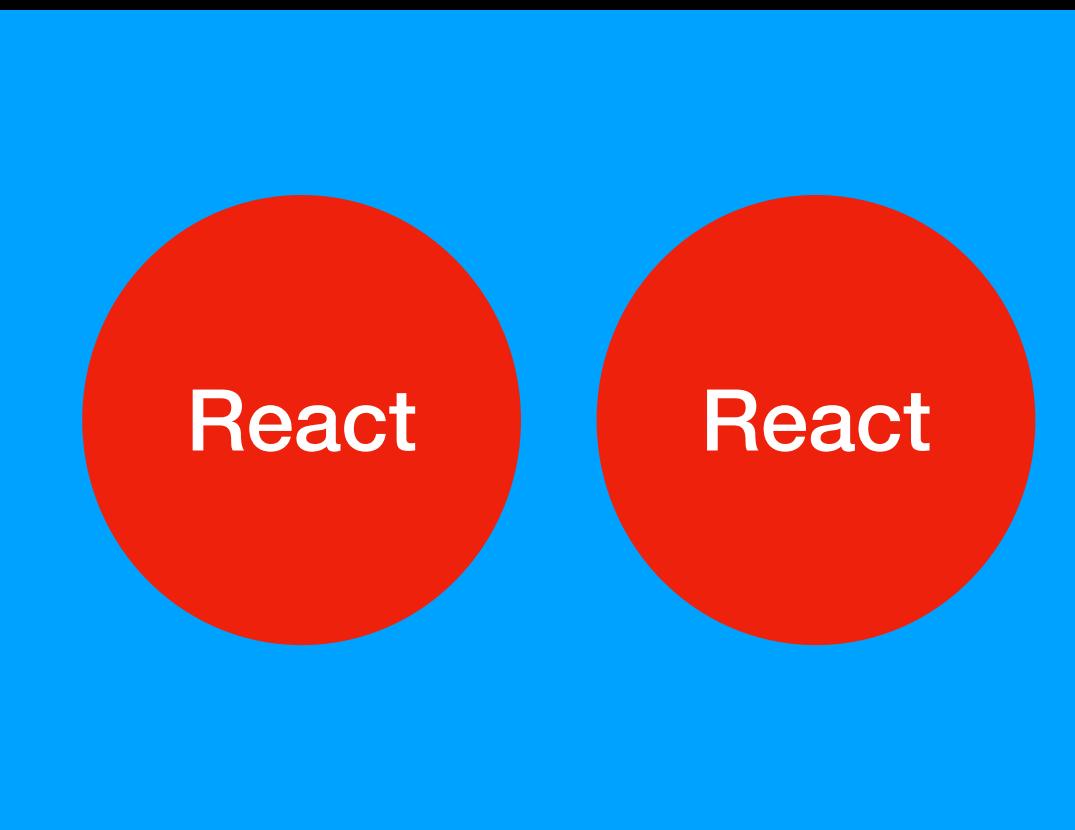


Container Orchestration - Kubernetes

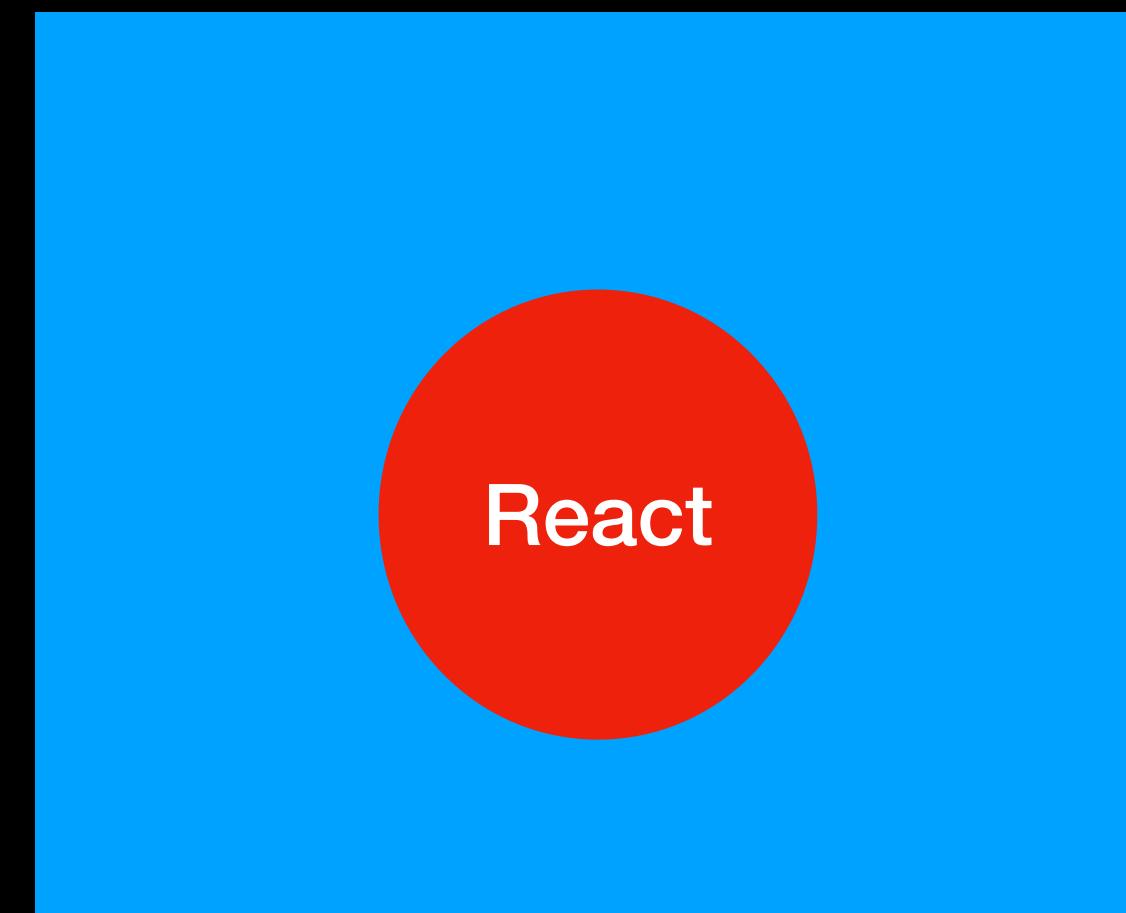
**Not just a file, you could
add and remove containers
by doing a function call**



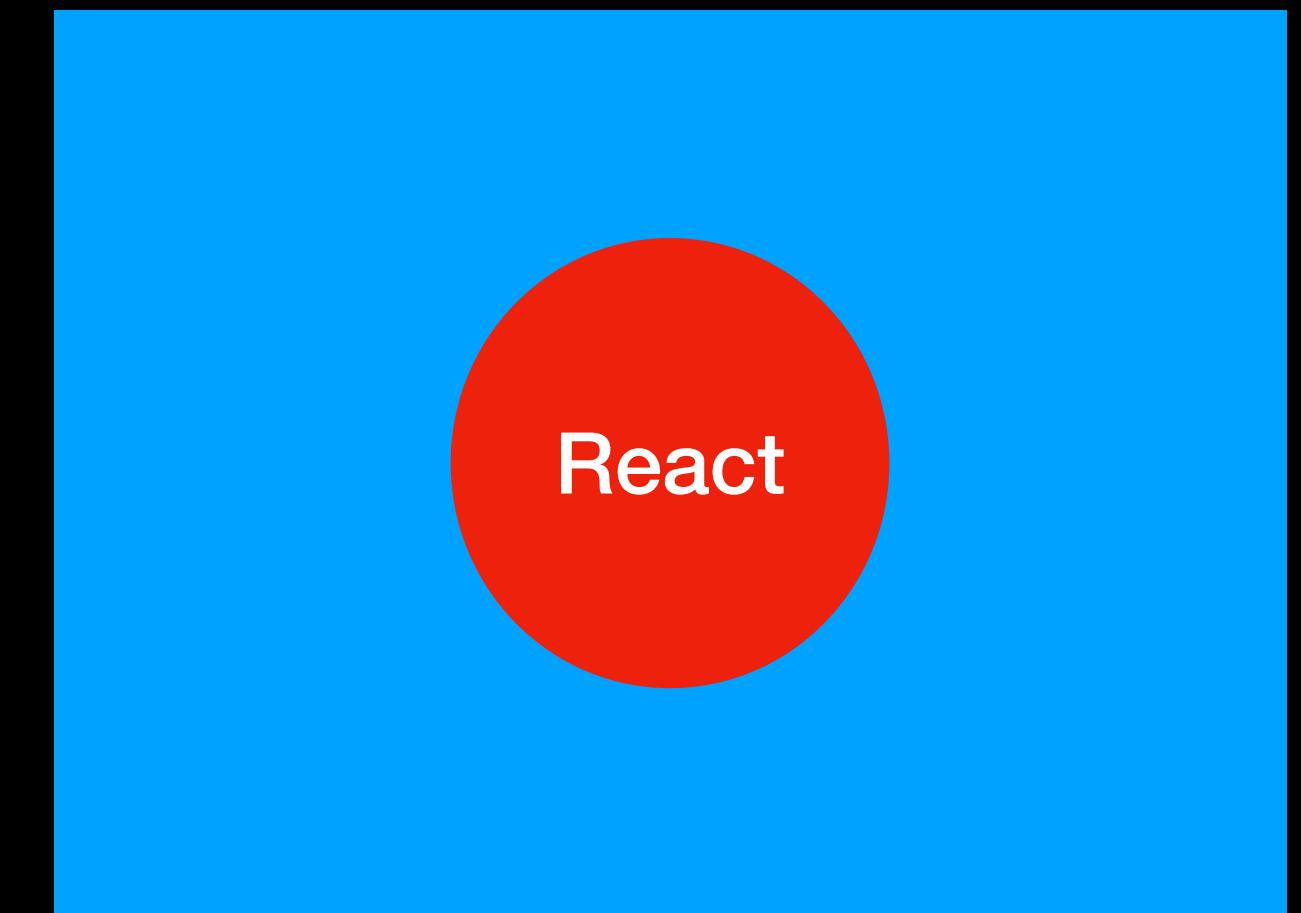
Mac



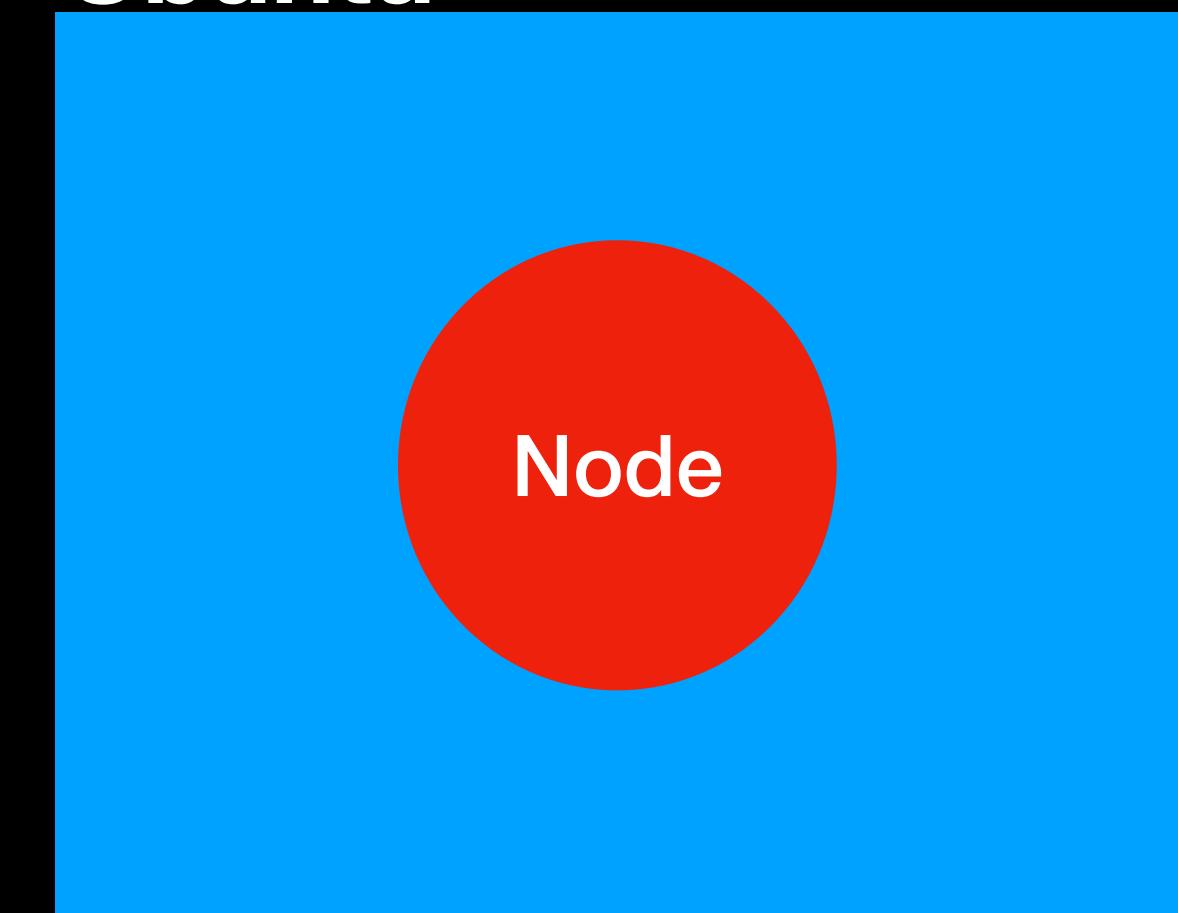
Windows



Linux



Ubuntu



Container Orchestration - Kubernetes

Some Kubernetes jargon

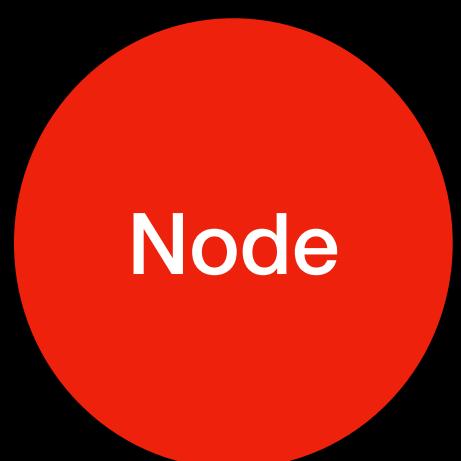
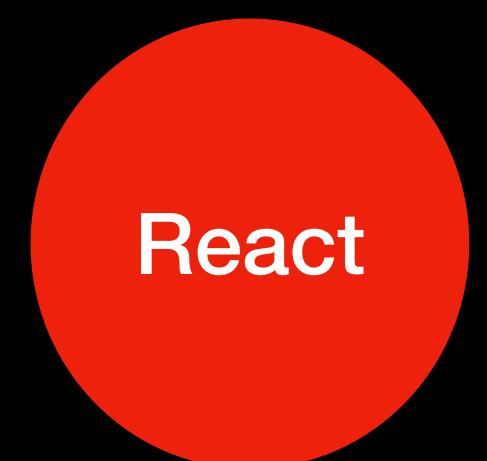
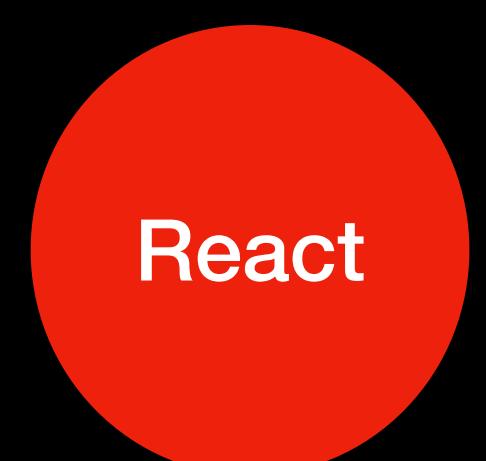
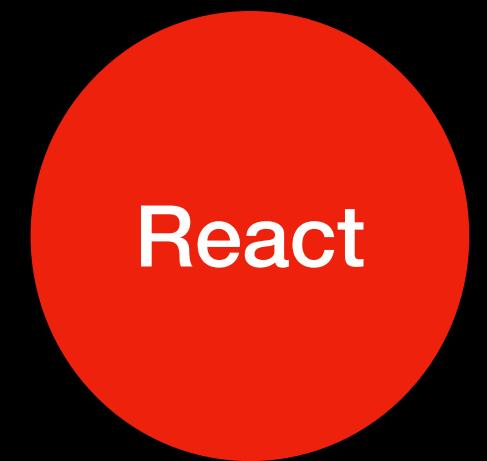
1. Nodes

Container Orchestration - Kubernetes

Some Kubernetes jargon

1. Nodes

2. Pods



Container Orchestration - Kubernetes

Some Kubernetes jargon

1. Nodes

2. Pods

3. Services

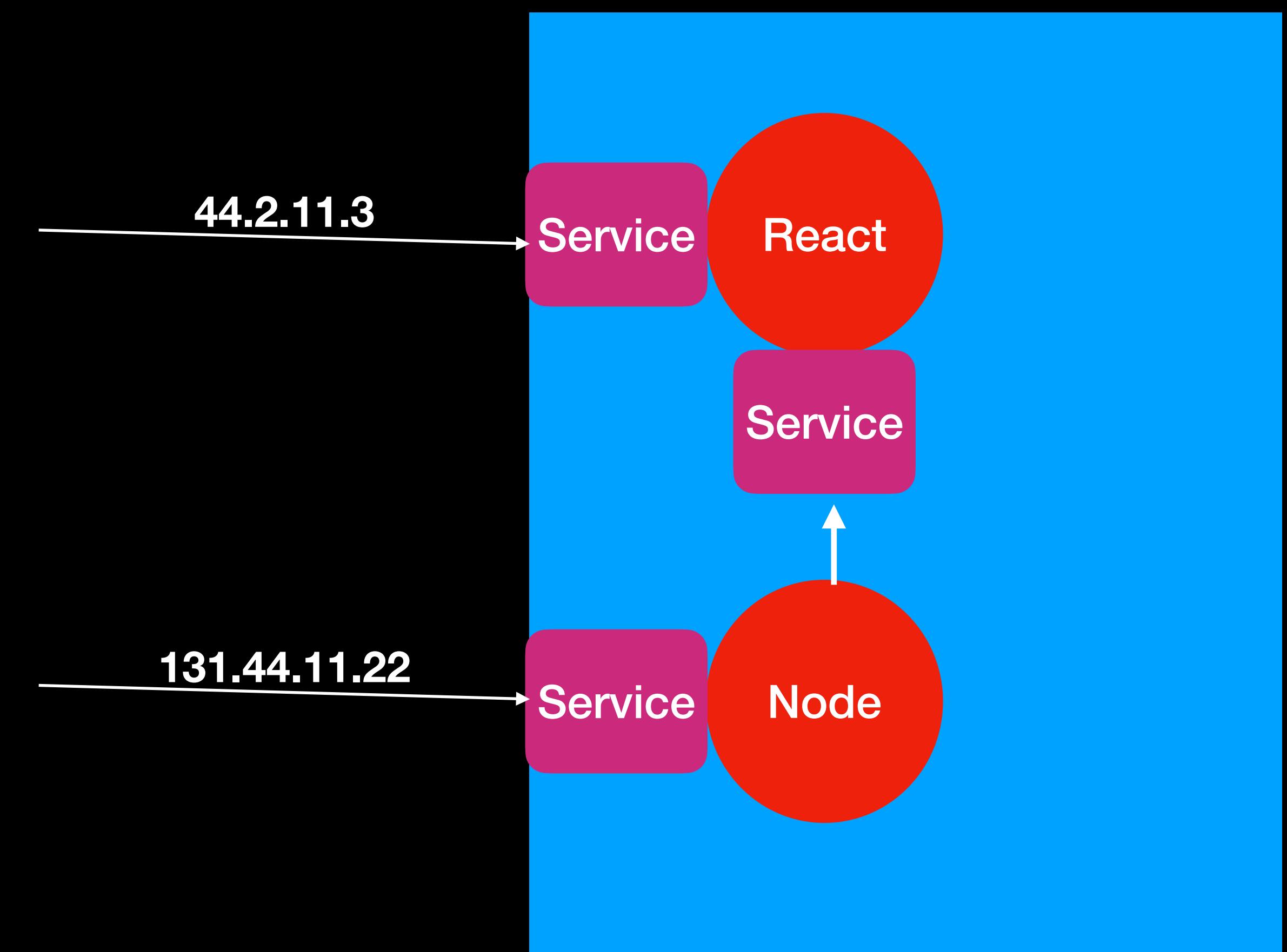
4. Ingress

Container Orchestration - Kubernetes

Services and ingress control how your application is exposed
How can people access your services running inside pods?
**How does a user that has created a repl, reach the container
where their code is present?**

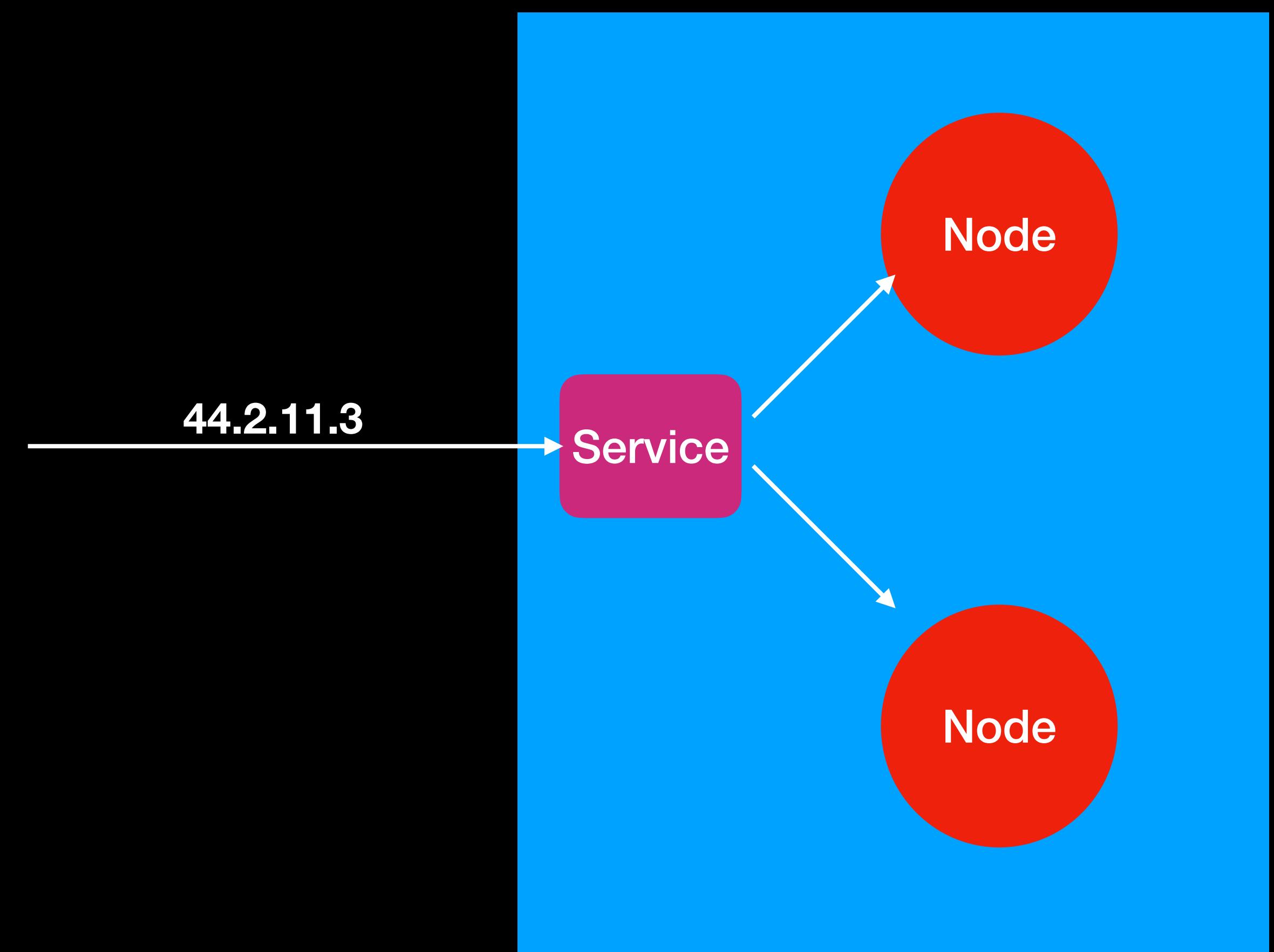
Container Orchestration - Kubernetes

- Services let you expose your pods**
1. You can either expose them on the internet
 2. Or you can let other people access your pods
 3. Each pod can have an associated service



Container Orchestration - Kubernetes

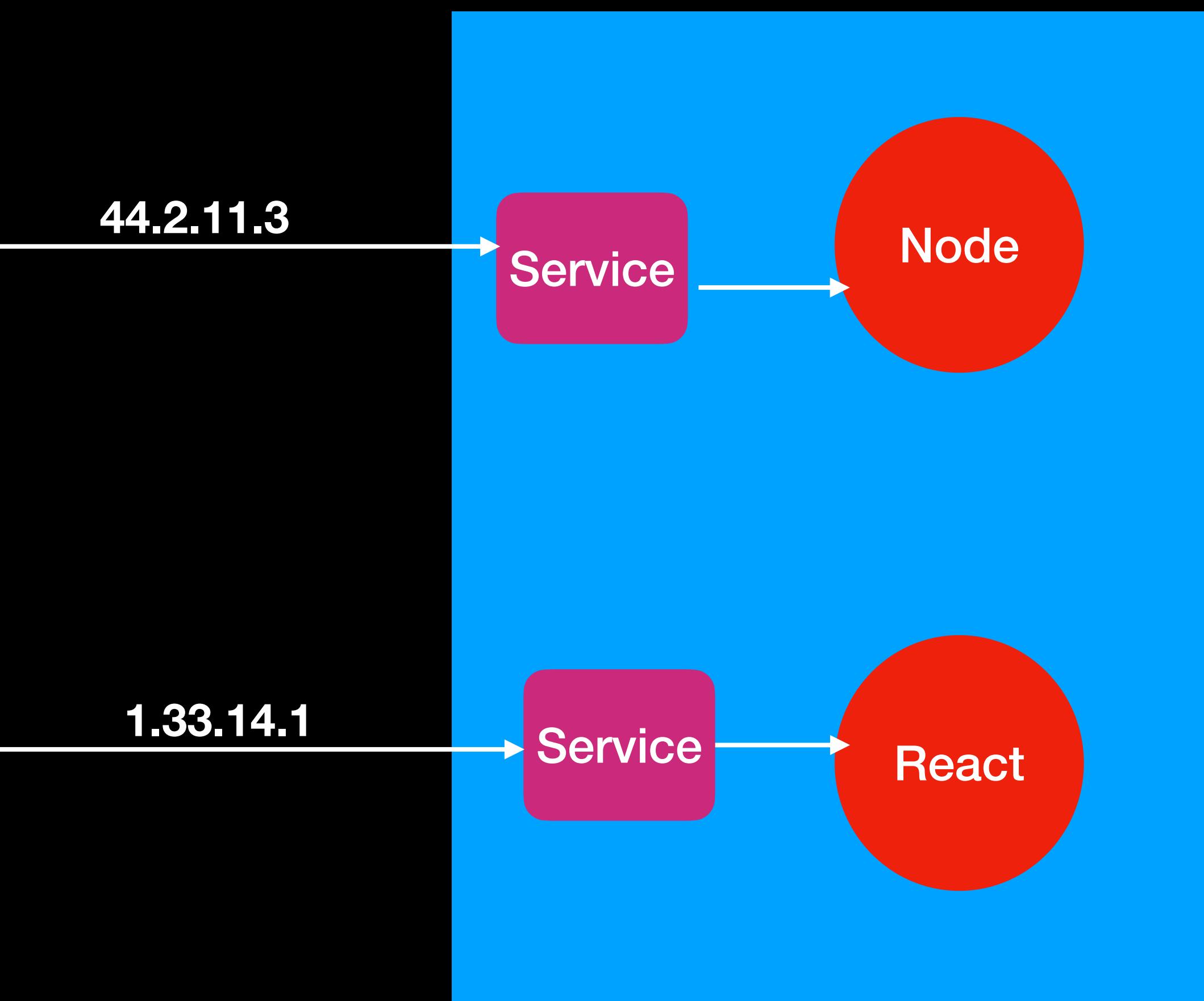
A service can also load balance across pods



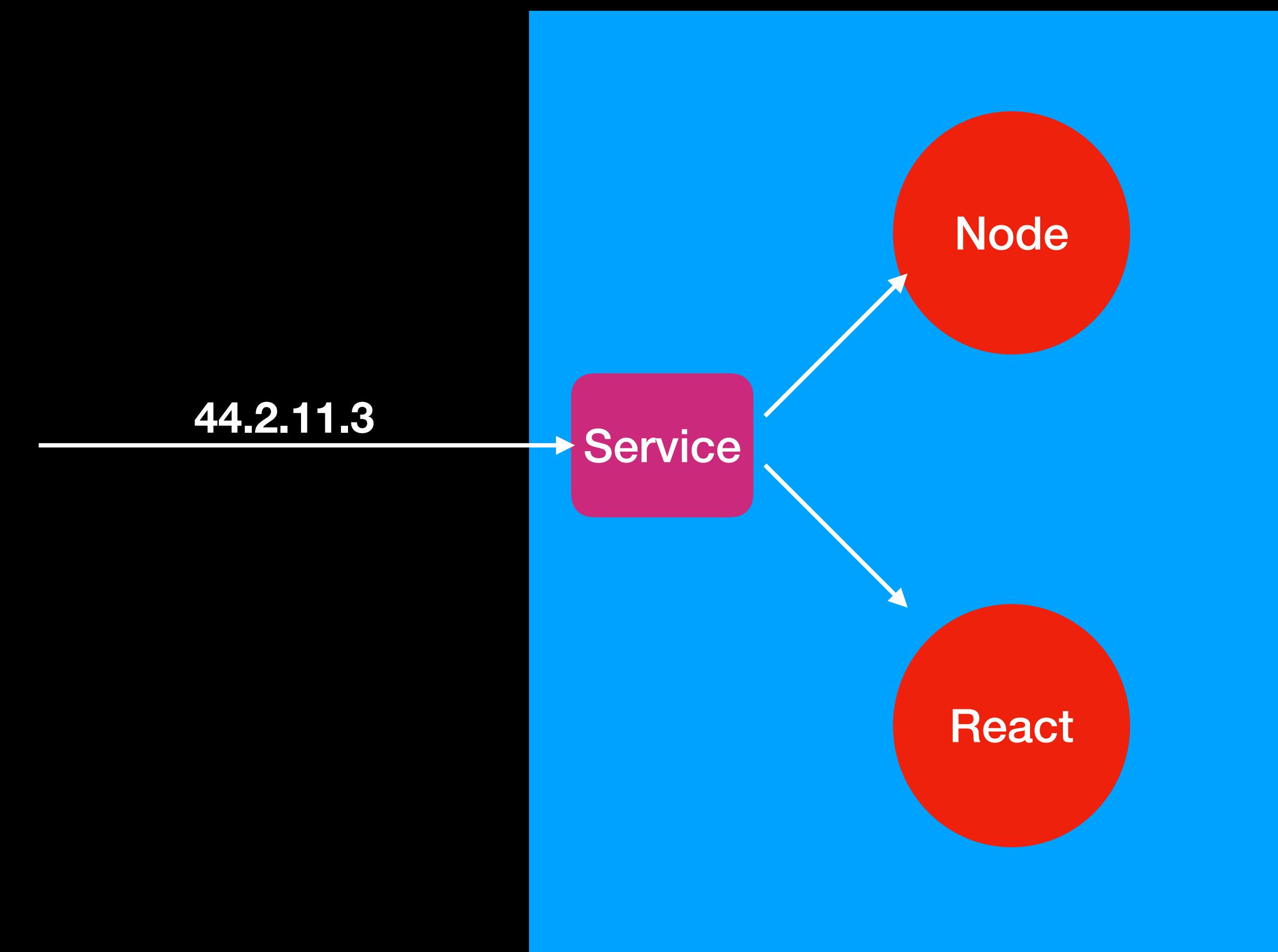
Container Orchestration - Kubernetes

Let's go through the replit runner and see
how they expose services
What do you think they do?

Approach #1



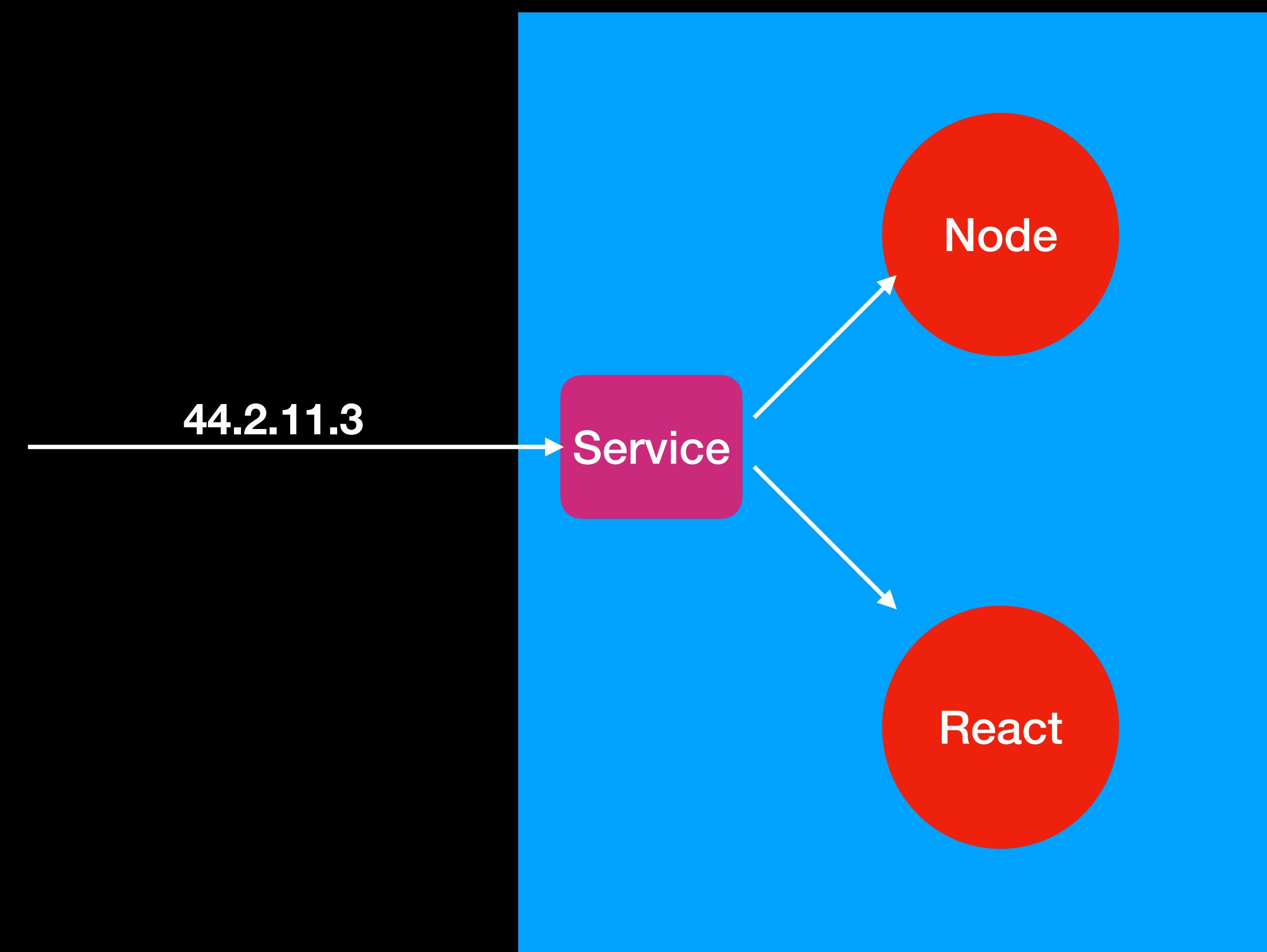
Approach #2



Container Orchestration - Kubernetes

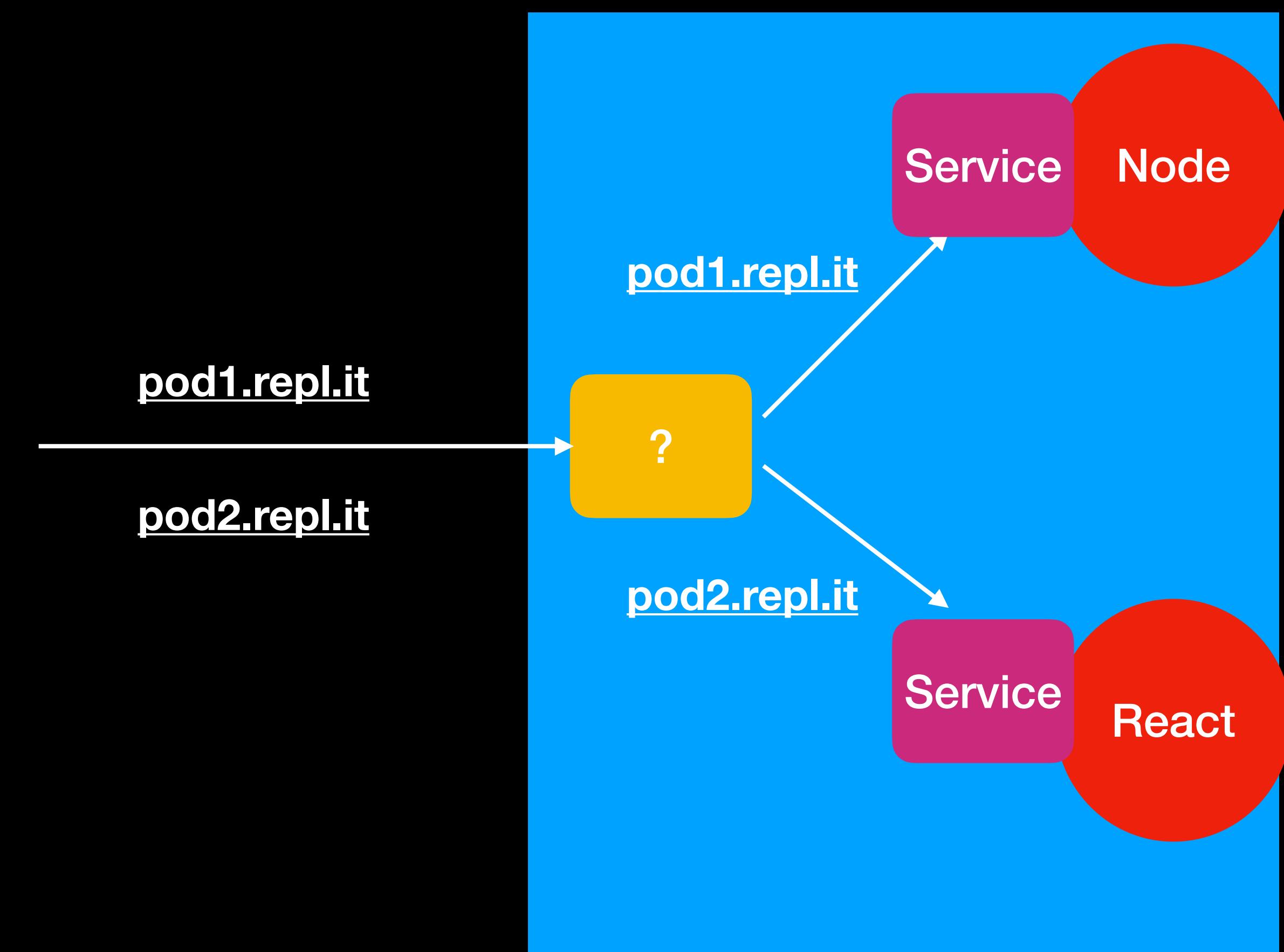
You might feel like they do this , and they do
But services don't let you do path based routing

They will load balance, but they won't let
you control where the request should go
based on the host URL



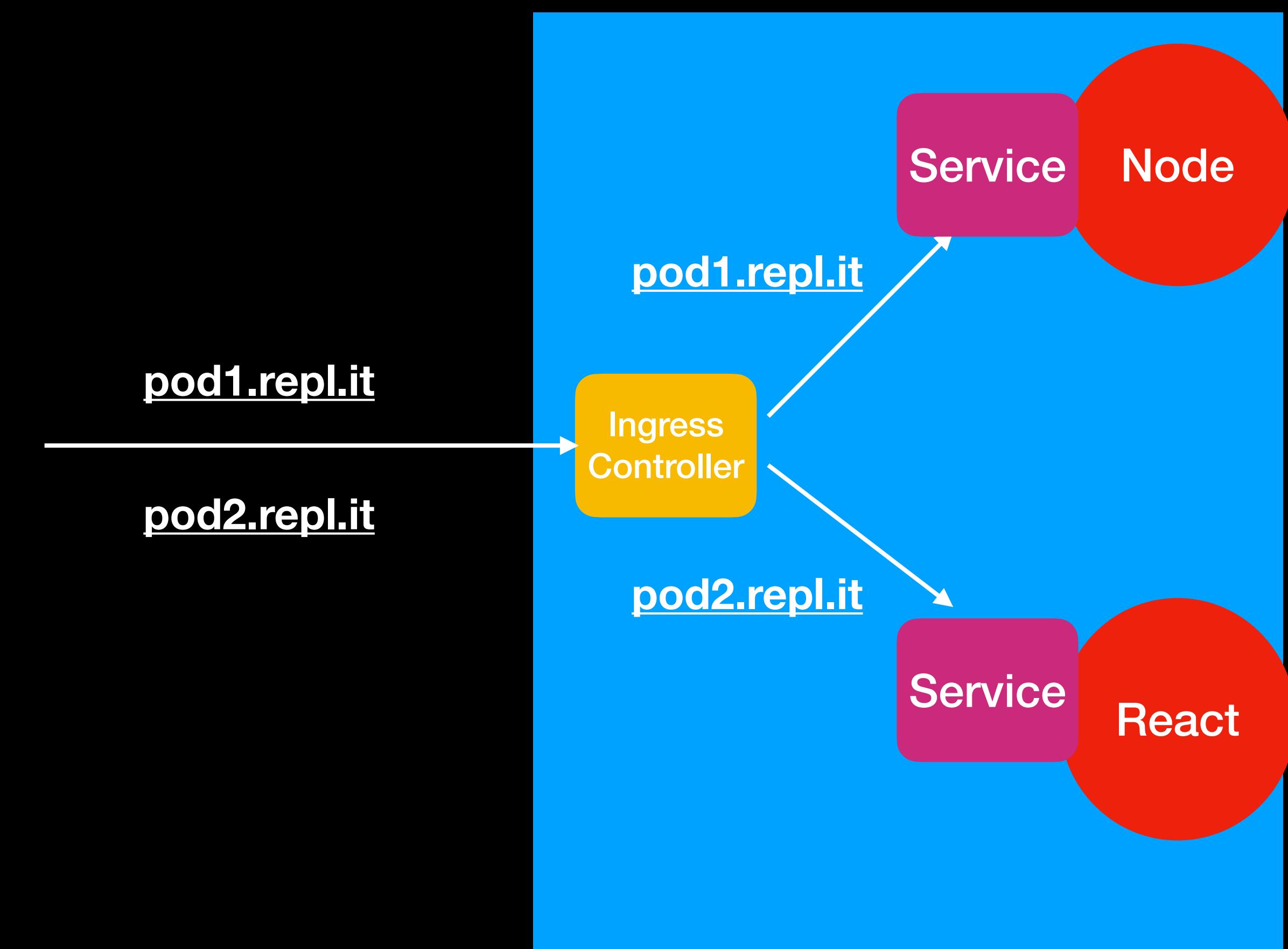
Container Orchestration - Kubernetes

How can we do path based routing?



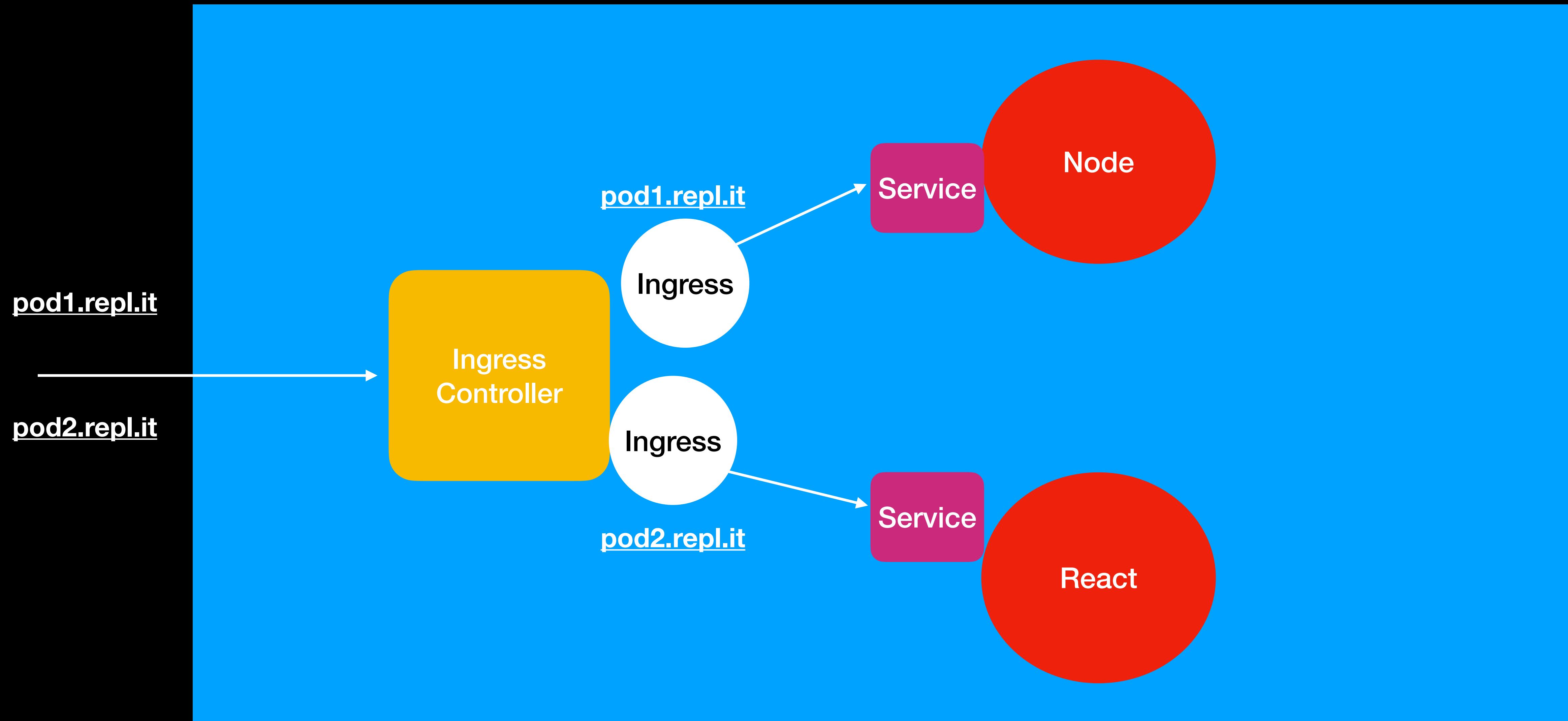
Container Orchestration - Kubernetes

Ingress



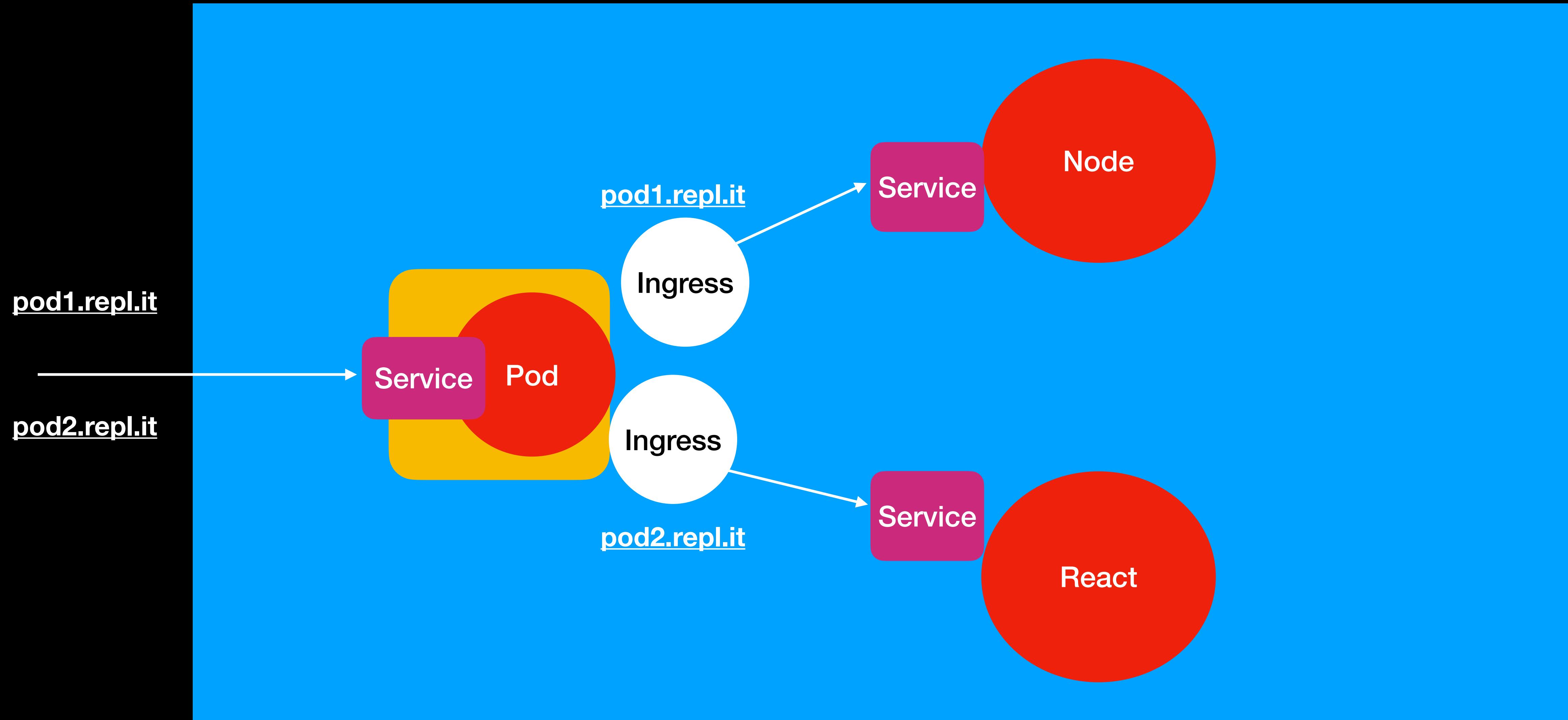
Container Orchestration - Kubernetes

Ingress Controller and Ingress



Container Orchestration - Kubernetes

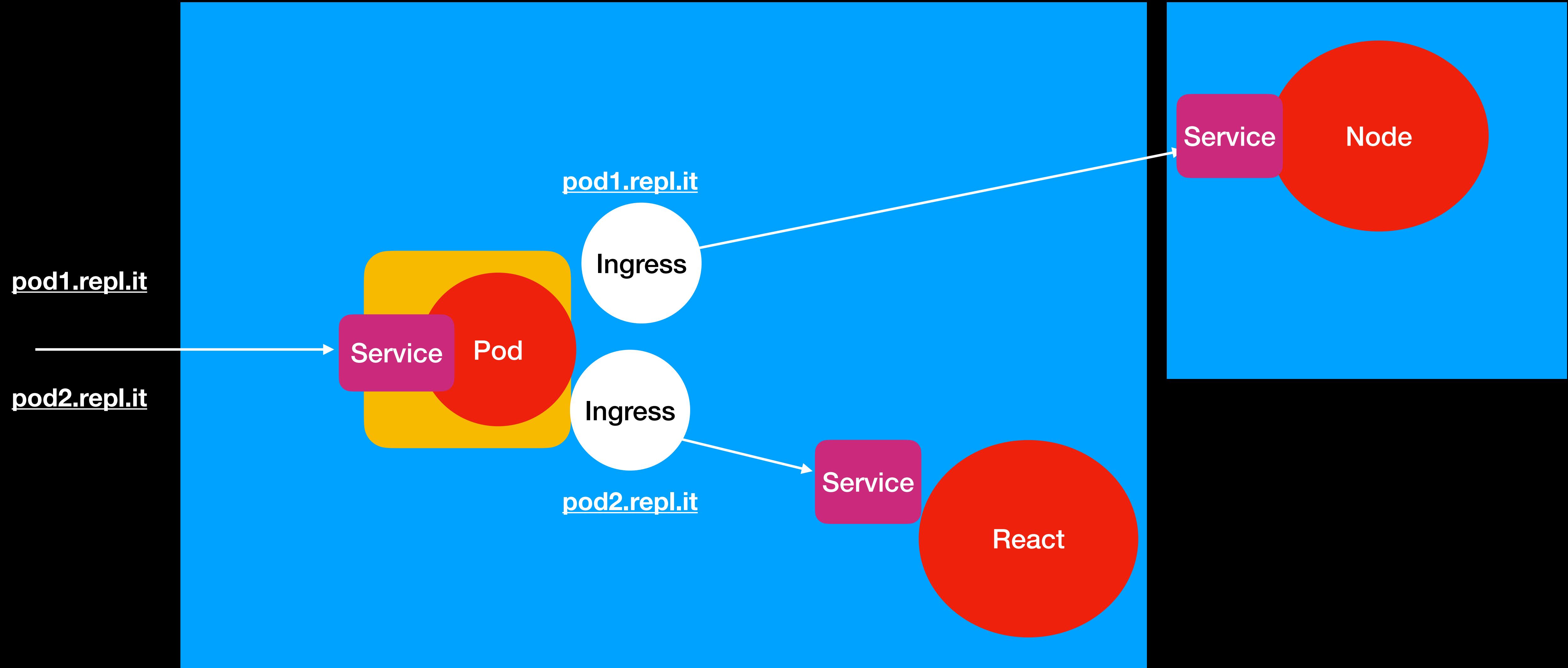
Ingress Controller and Ingress



Container Orchestration - Kubernetes

They can exist on different nodes as well

As long as they are in the same cluster, Ingress controller should be able to route traffic

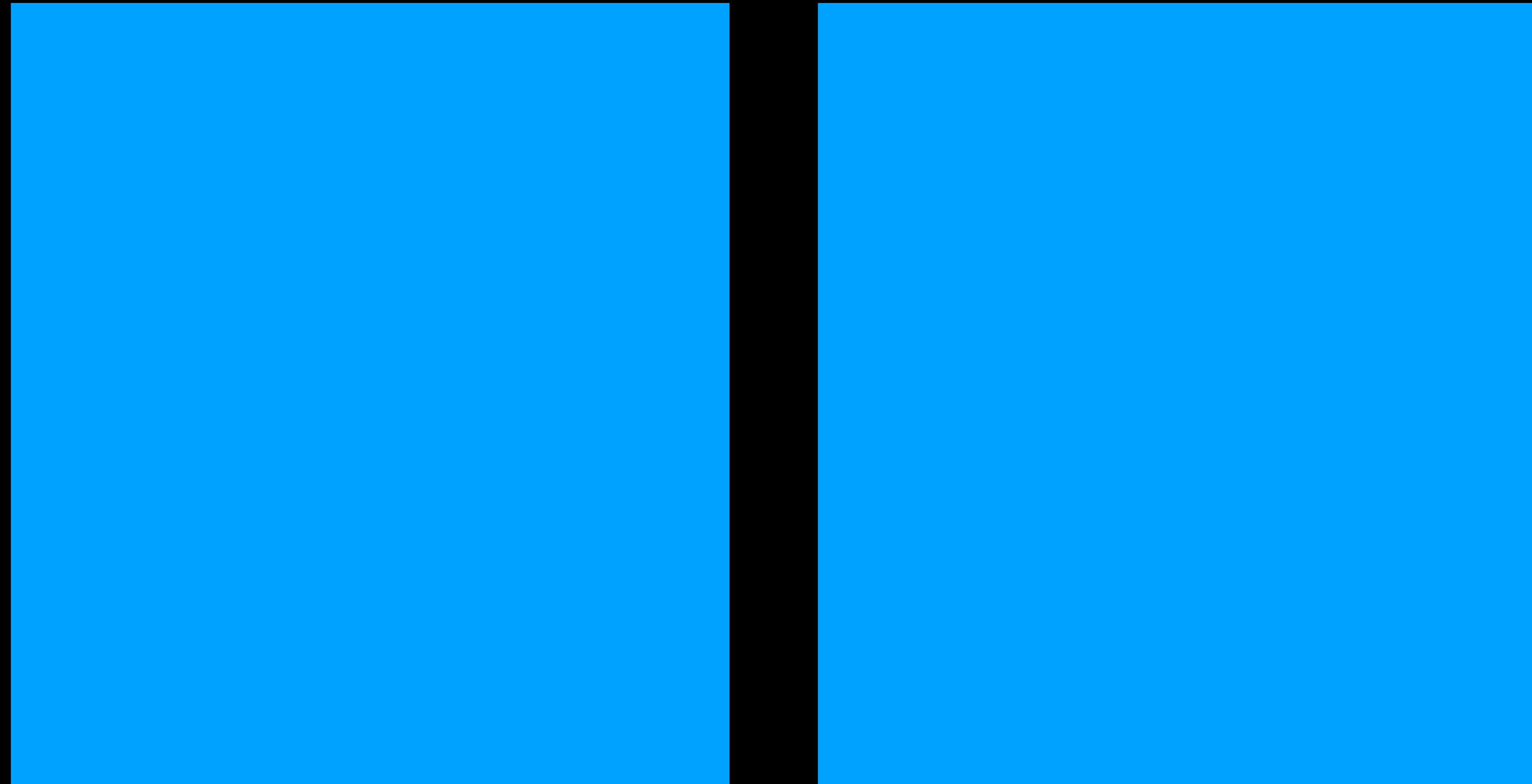


Container Orchestration - Kubernetes

Given all this information, can you guess the final architecture?

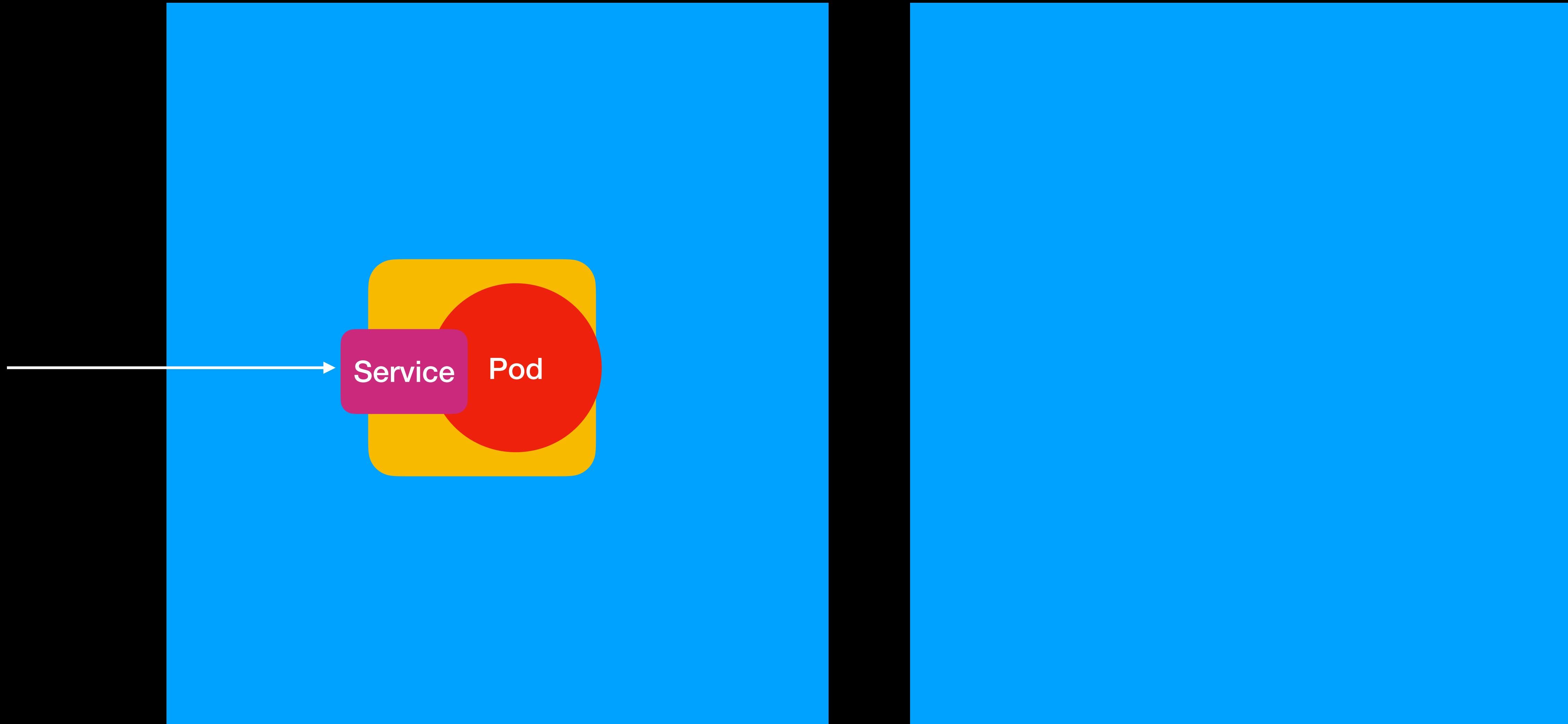
Container Orchestration - Kubernetes

Step 1 - Start a k8s cluster, set some autoscaling policies on the nodes



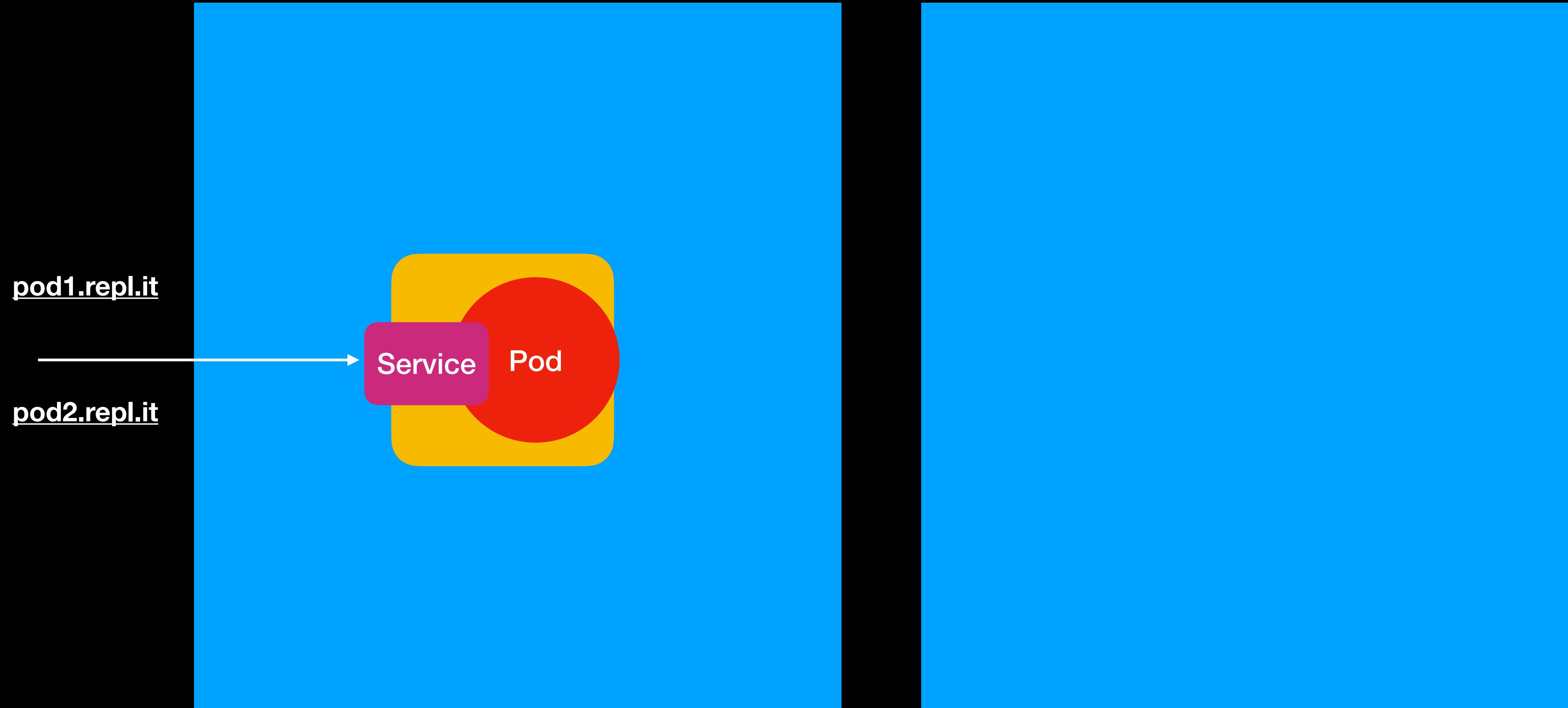
Container Orchestration - Kubernetes

Step 2 - Attach an ingress controller to the cluster (one time)



Container Orchestration - Kubernetes

Step 3 - Point your DNS to the IP of the ingress controller



Container Orchestration - Kubernetes

Step 3 - Point your DNS to the IP of the ingress controller

The screenshot shows a DNS configuration page with a red box highlighting the custom record entry.

Custom records
*.gptflock.com/A

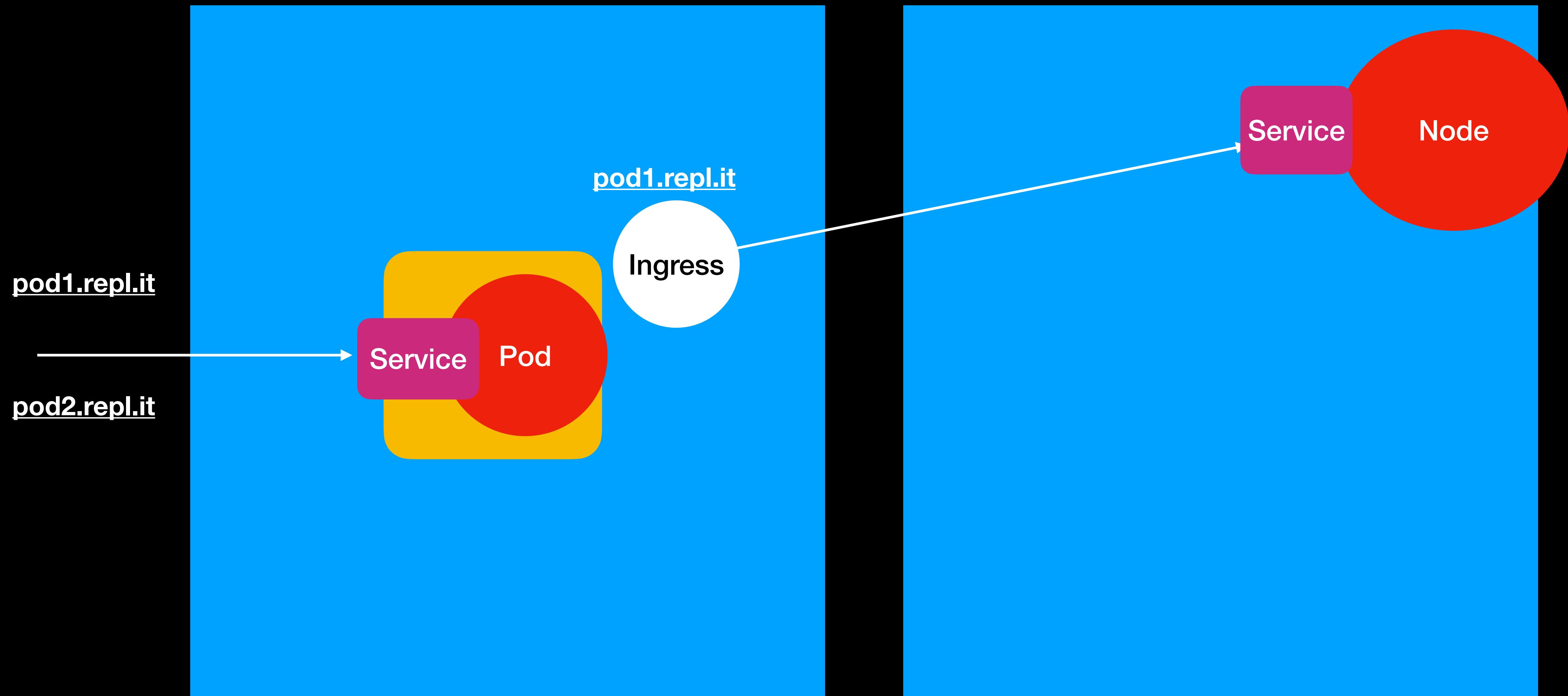
[Manage custom records](#)

Host name	Type	TTL	Data
*.gptflock.com	A	1 hour	139.144.255.233

Required
Informational-only records

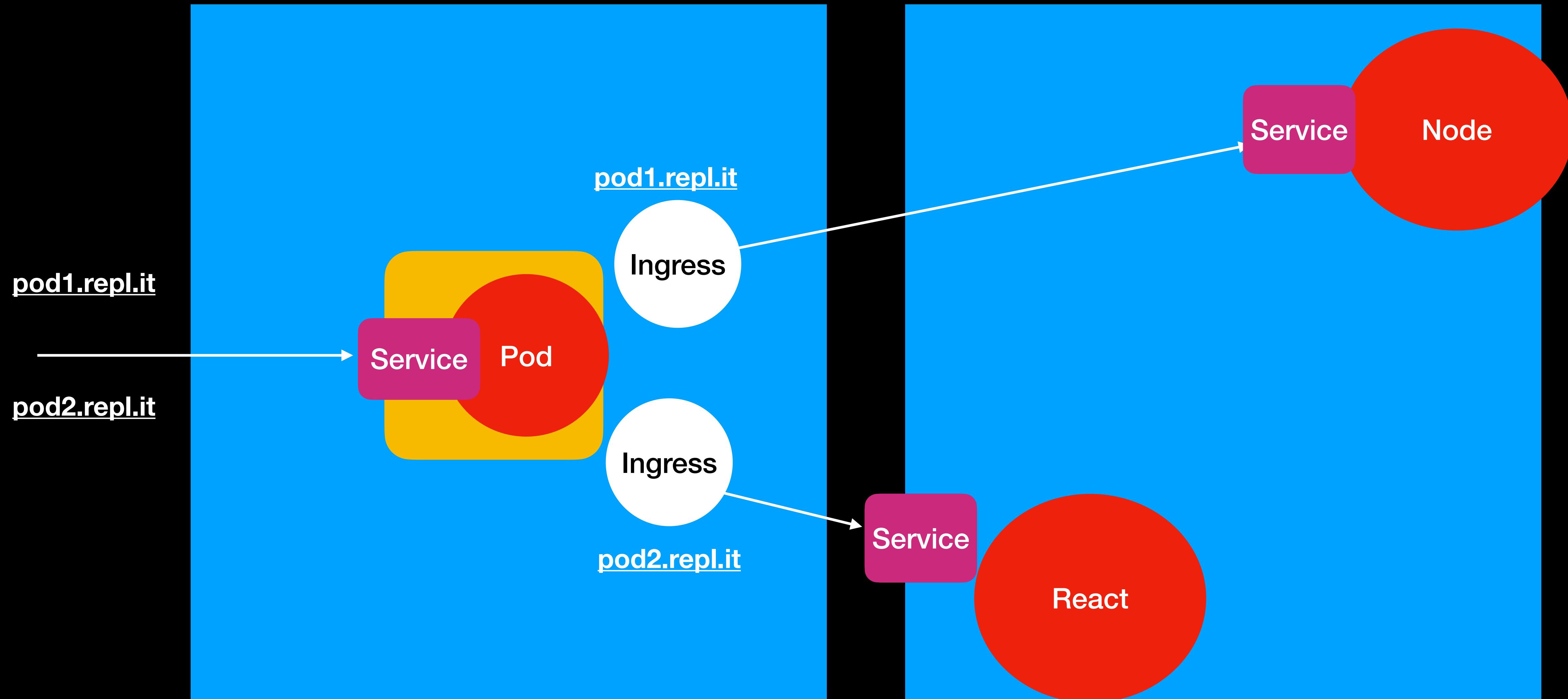
Container Orchestration - Kubernetes

Step 4 - As people start repls, start a pod, service and ingress for them



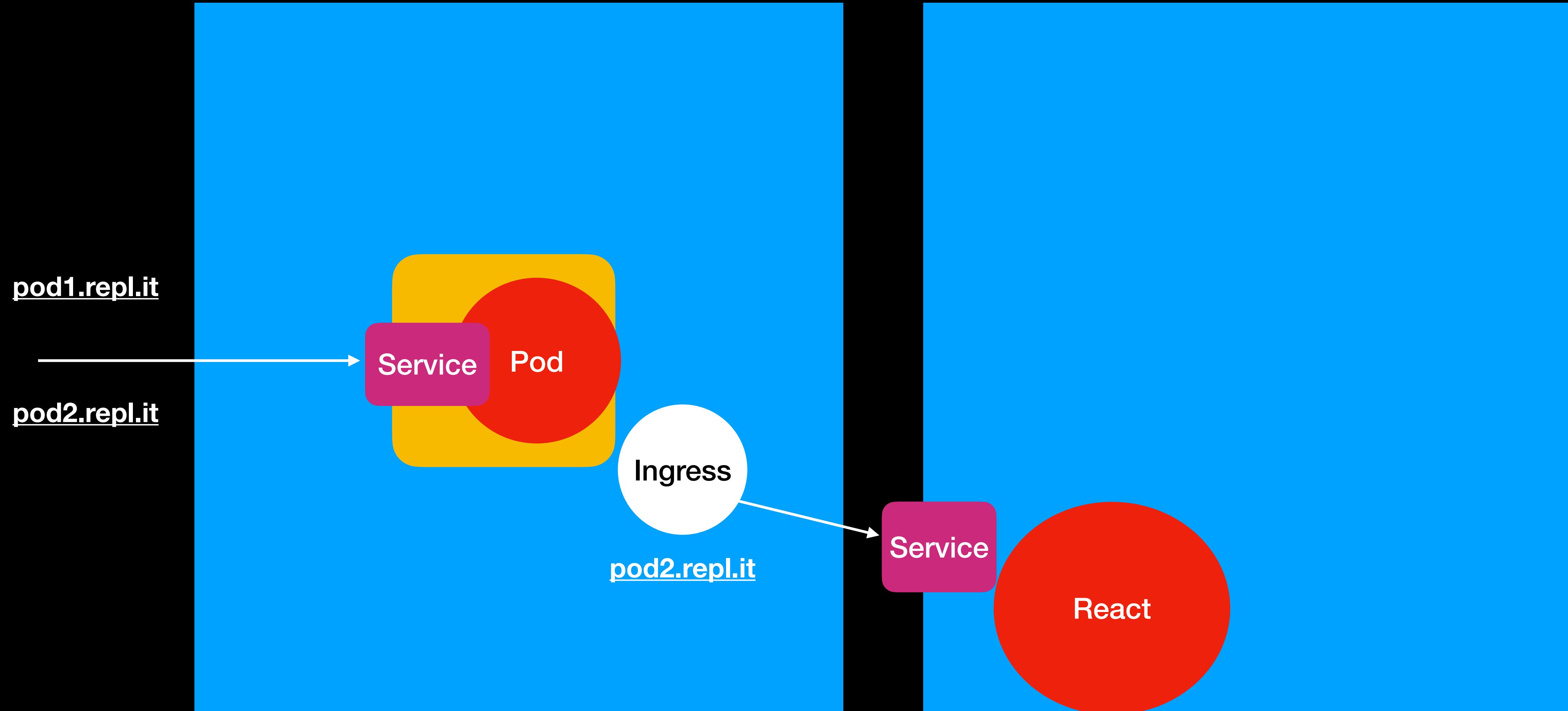
Container Orchestration - Kubernetes

Step 4 - As people start repls, start a pod, service and ingress for them



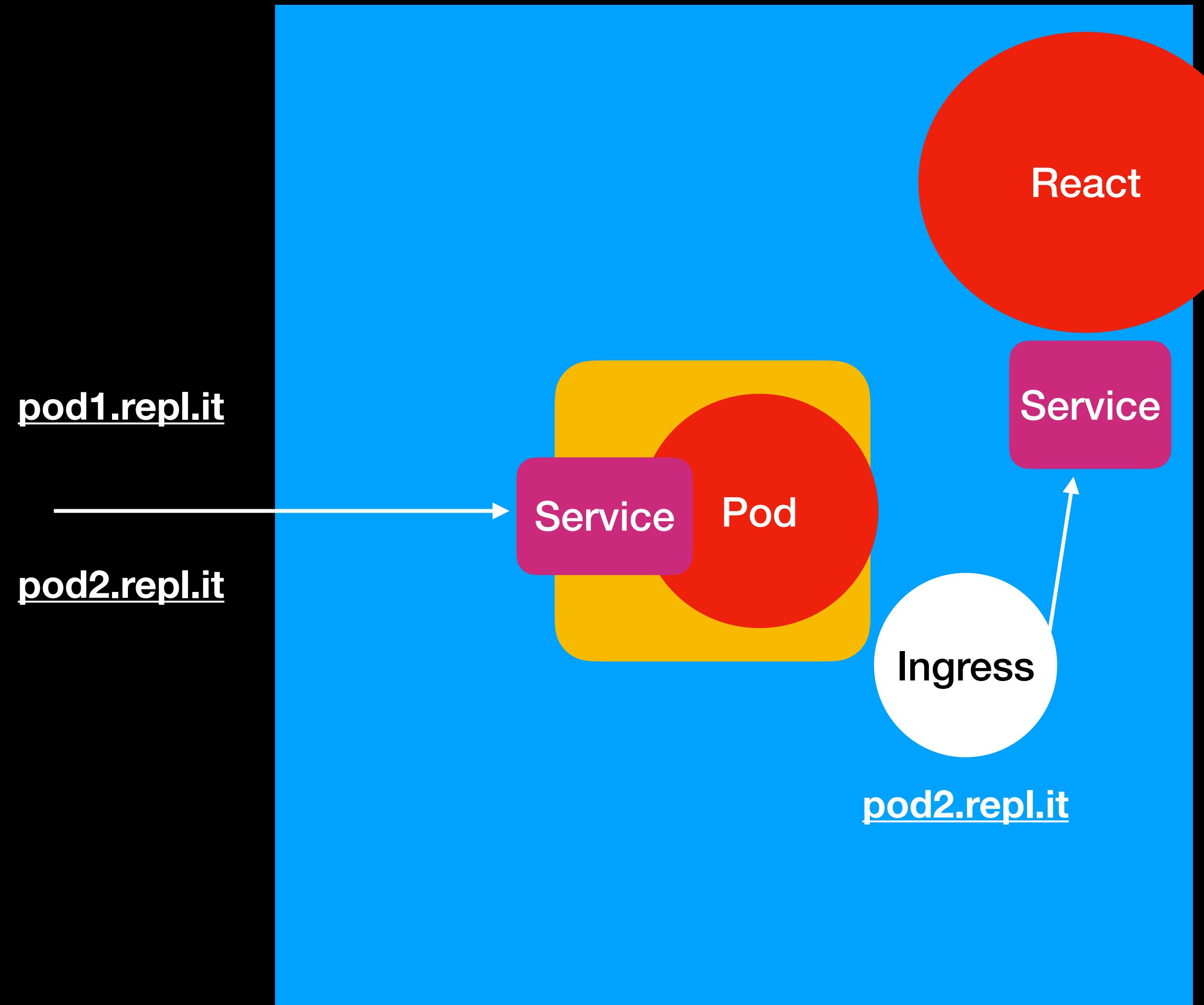
Container Orchestration - Kubernetes

Step 4 - As people leave repls, stop the respective pod, service and ingress



Container Orchestration - Kubernetes

Step 4 - Cluster will autoscale based on the policies you added



Container Orchestration - Kubernetes

You don't NEED **Kubernetes** for what we're doing today
Kubernetes gives you a bunch of things that we don't need today (for example deployments, automatic restarts, recycling containers ...)

Part 2 | The good solution

Some more jargon

Basic Jargon

1. **Containers (Docker)**
2. **Container Orchestration (Kubernetes)**

Advance Jargon (not needed for this tutorial)

1. Reproducible builds - Nix
2. Network volumes
3. Caching dependencies

Part 2 | The good solution

My proposed solution

Part 2 | The good solution

We have 3 services, and a k8s cluster

Simple HTTP API

Orchestrator
(http or ws)

Runner ws server

32 cpus
100gb

32 cpus
100gb

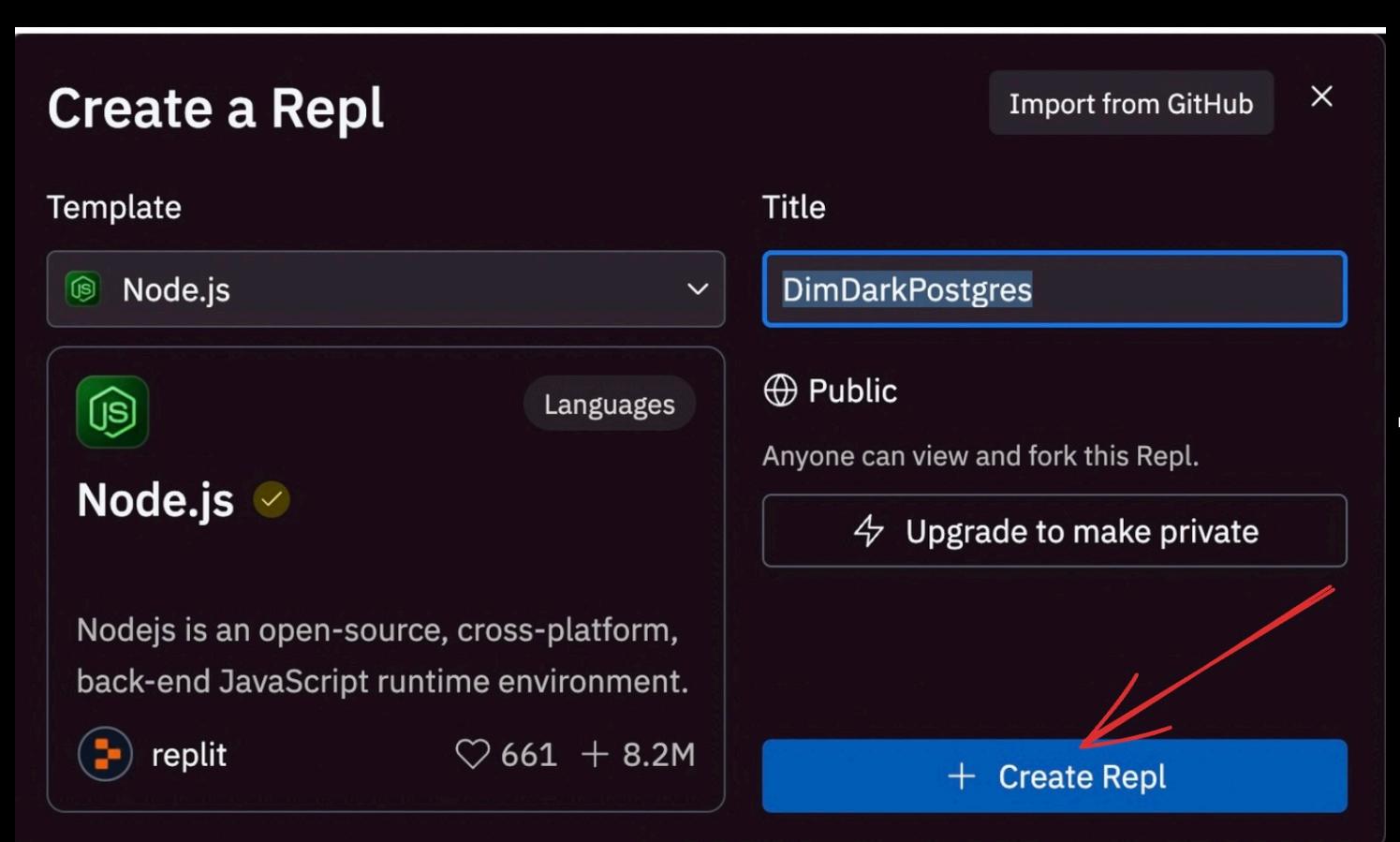
32 cpus
100gb

32 cpus
100gb

1. Simple HTTP API

Step 1 - Initialising the repl

Copy over the base image to
s3://images/{id}



Simple HTTP API

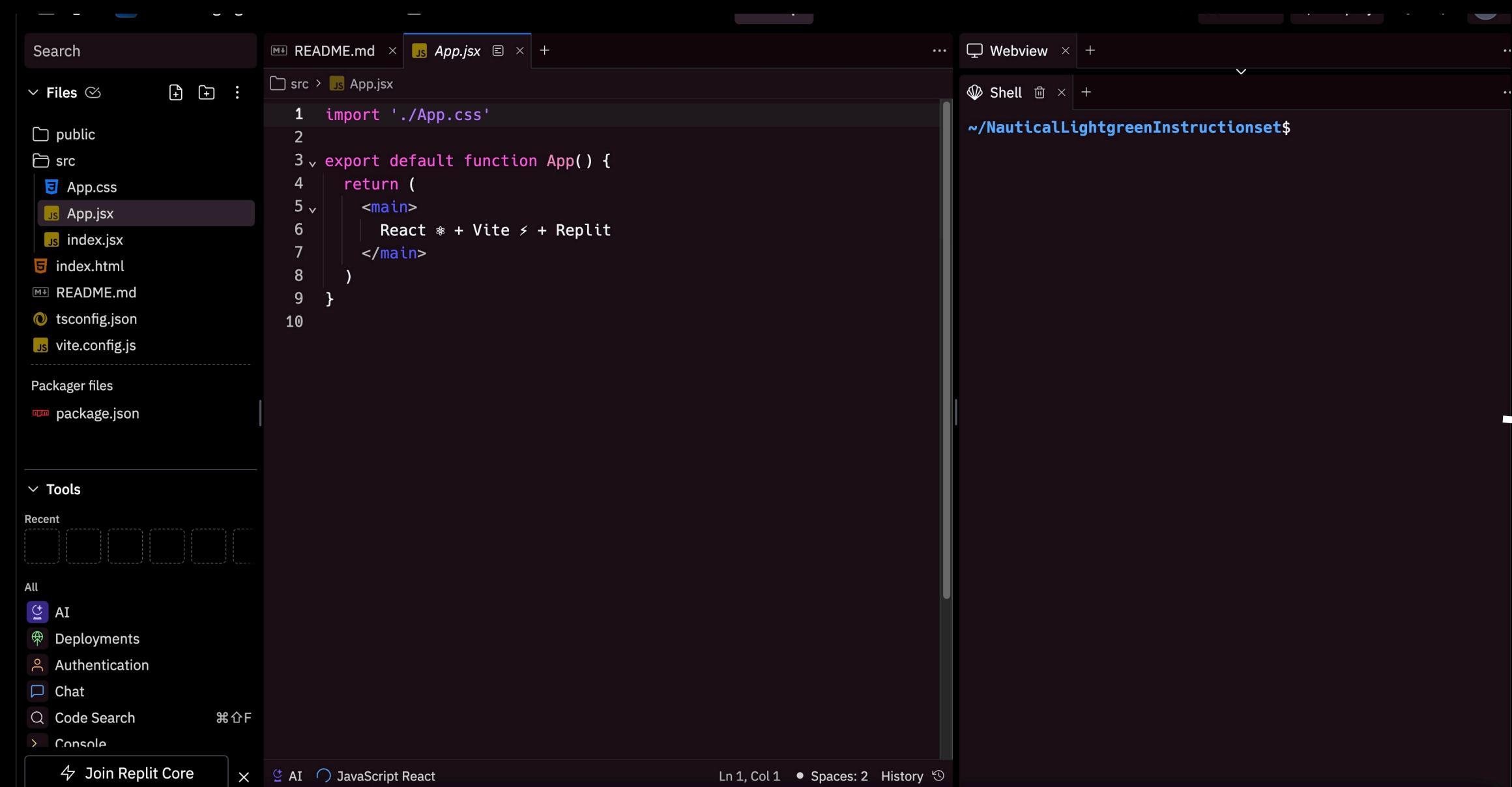
The screenshot shows the AWS S3 console. A red box highlights the 'base-node-code/' folder in the list of buckets. A red arrow points from this selection to the 'DimDarkPostgres/' folder listed below, which contains two objects: 'index.js' and 'package.json'.

Name	Type
base-go-code/	Folder
base-node-code/	Folder
base-react-code/	Folder
base-rust-code/	Folder

Name	Type
index.js	js
package.json	json

2. Runner Service

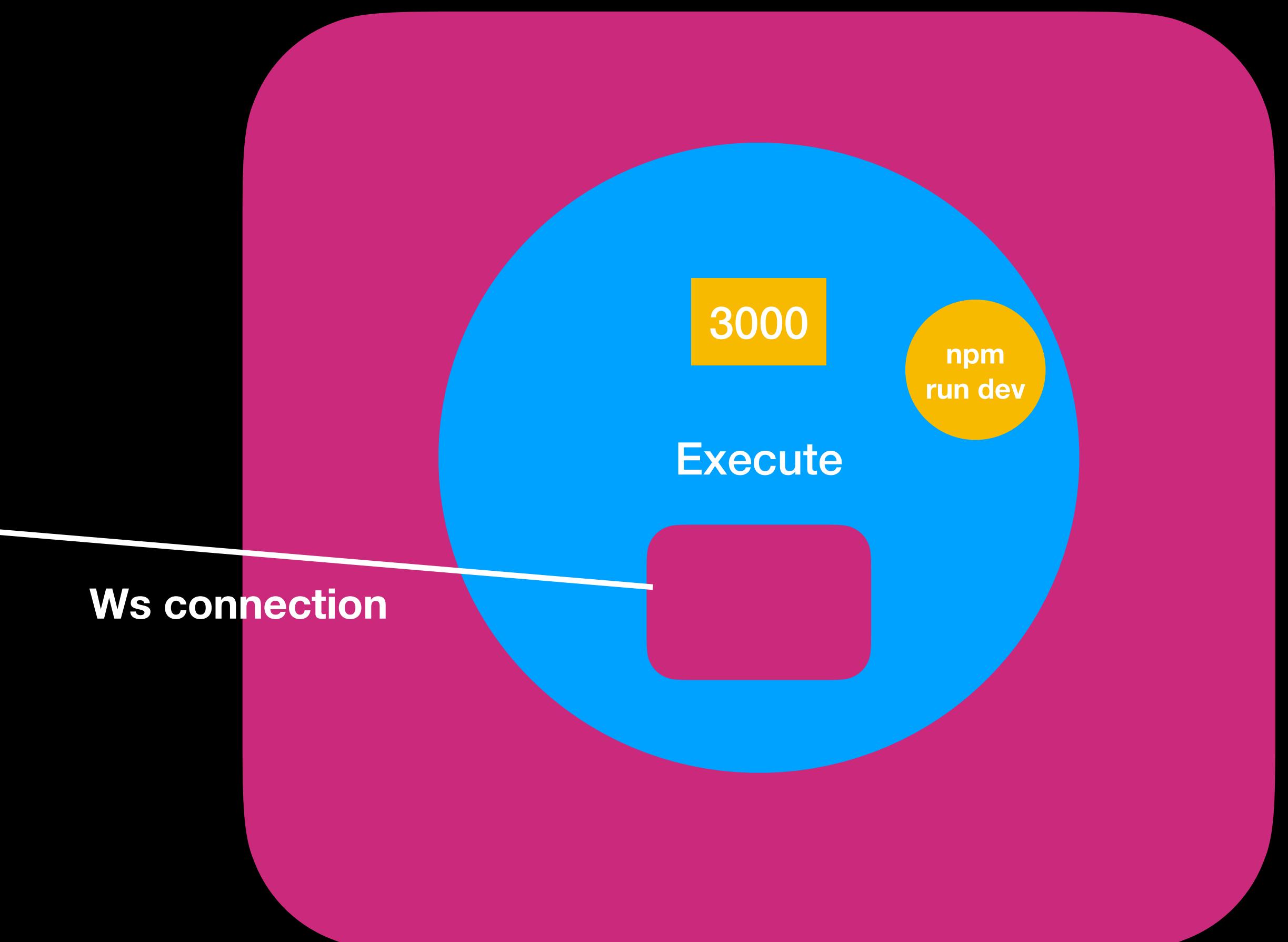
This is the same as the web socket service we built in the last section



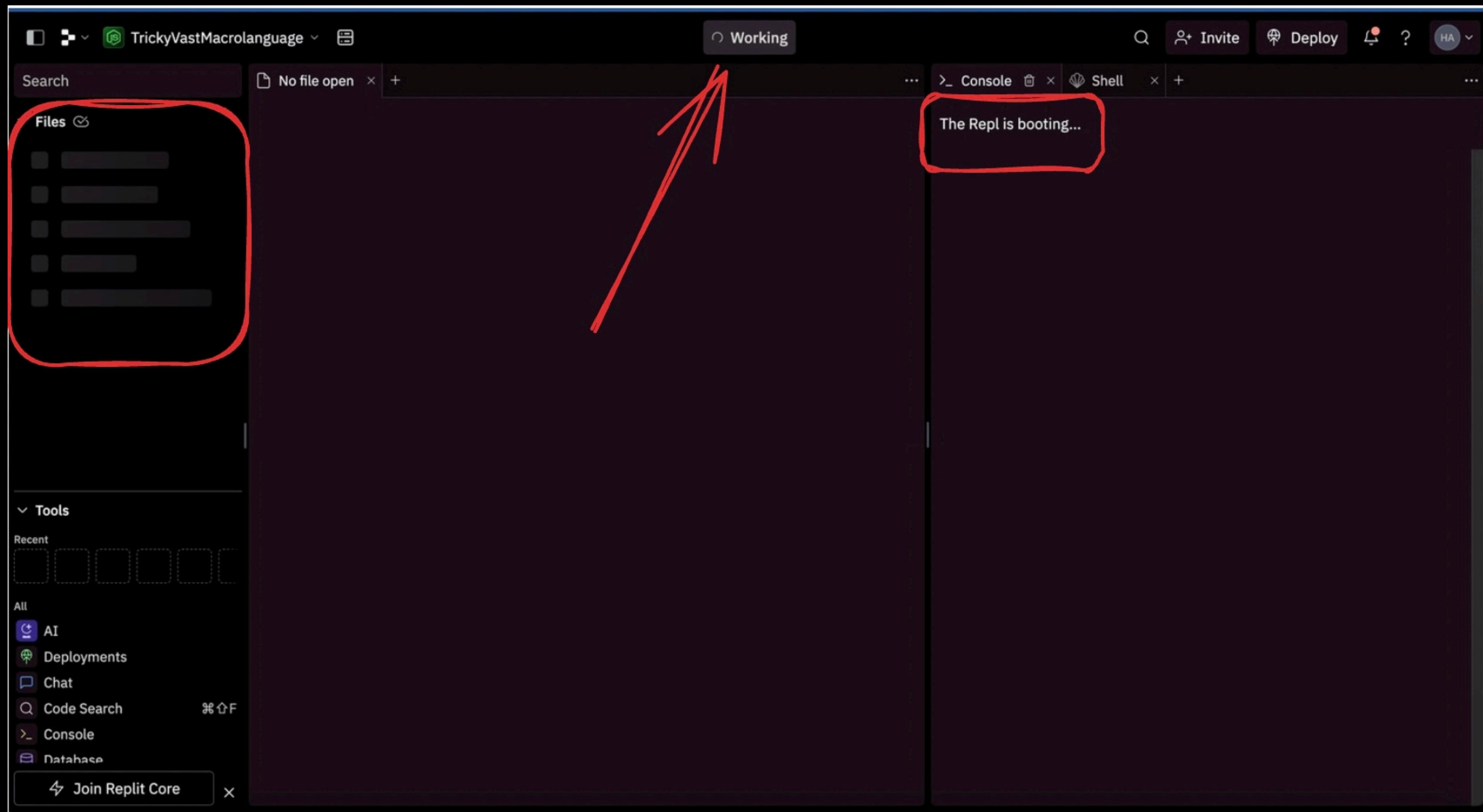
The screenshot shows the Replit IDE interface. On the left, the file tree displays files like README.md, App.css, App.jsx (selected), index.jsx, index.html, and tsconfig.json. The main workspace shows code for App.jsx:

```
1 import './App.css'
2
3 export default function App() {
4   return (
5     <main>
6       React * + Vite * + Replit
7     </main>
8   )
9 }
```

Below the code editor is a shell terminal window showing the command `npm run dev`. At the bottom, there are tabs for AI, JavaScript React, and a console.

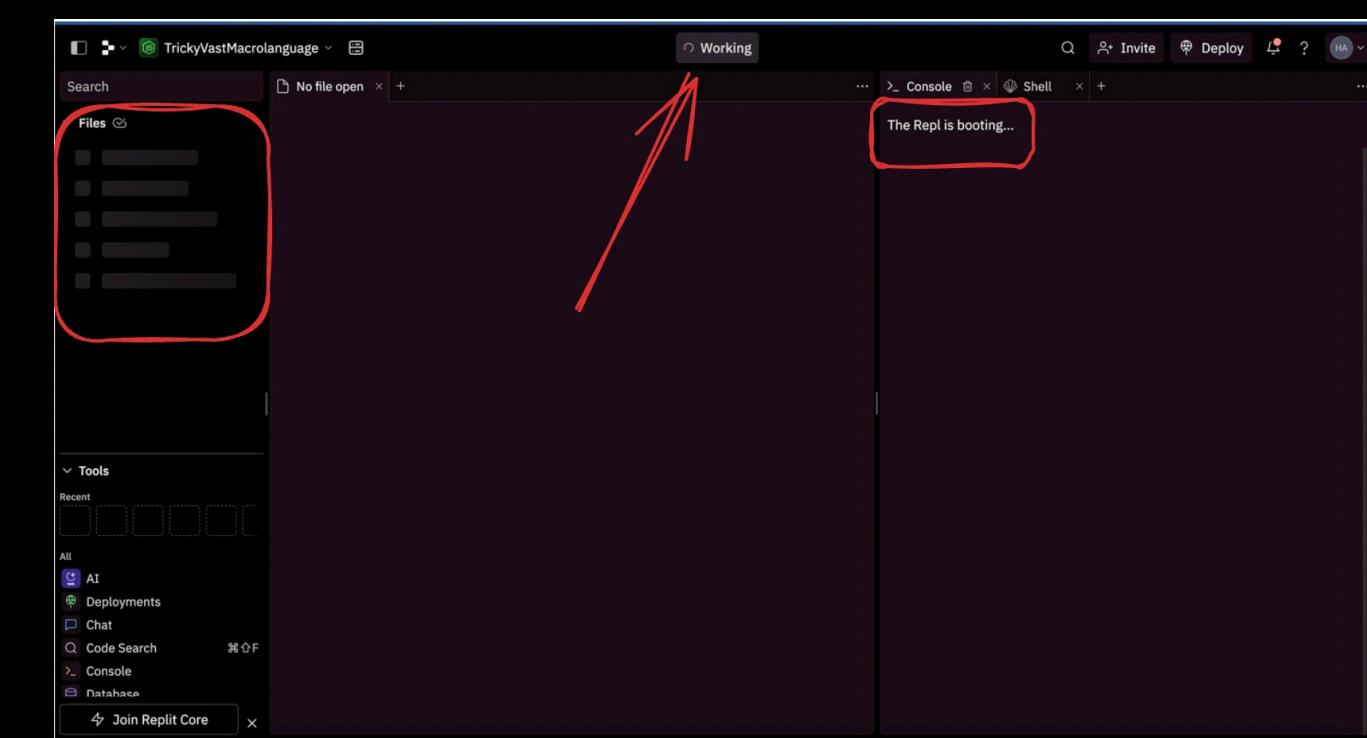


**What happens after the user starts the repl?
We need to start an independent runner for them
While it starts, the user sees the loading screen**

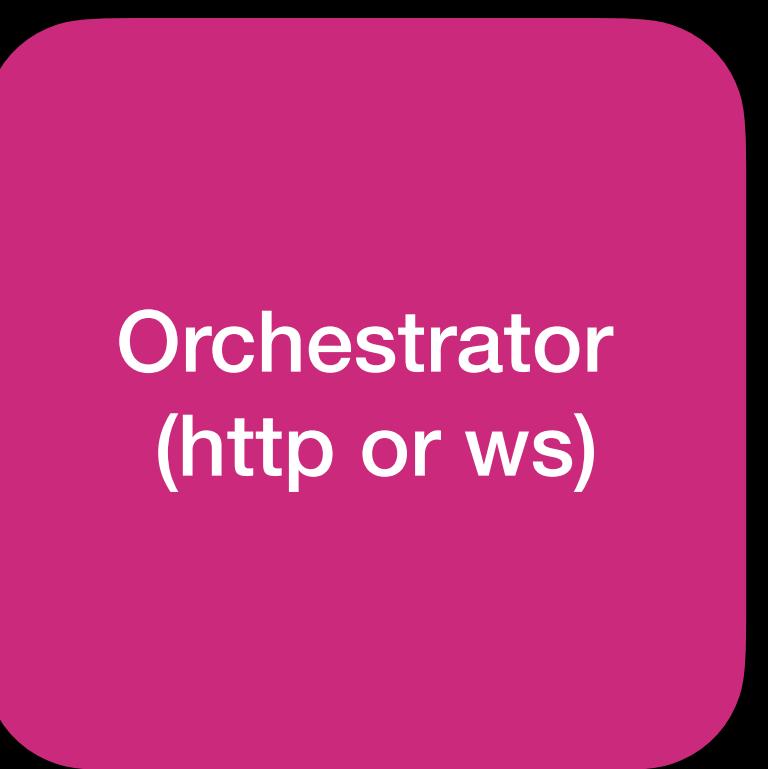


3. Orchestrator

Step 3 -Tell the orchestrator to start a pod

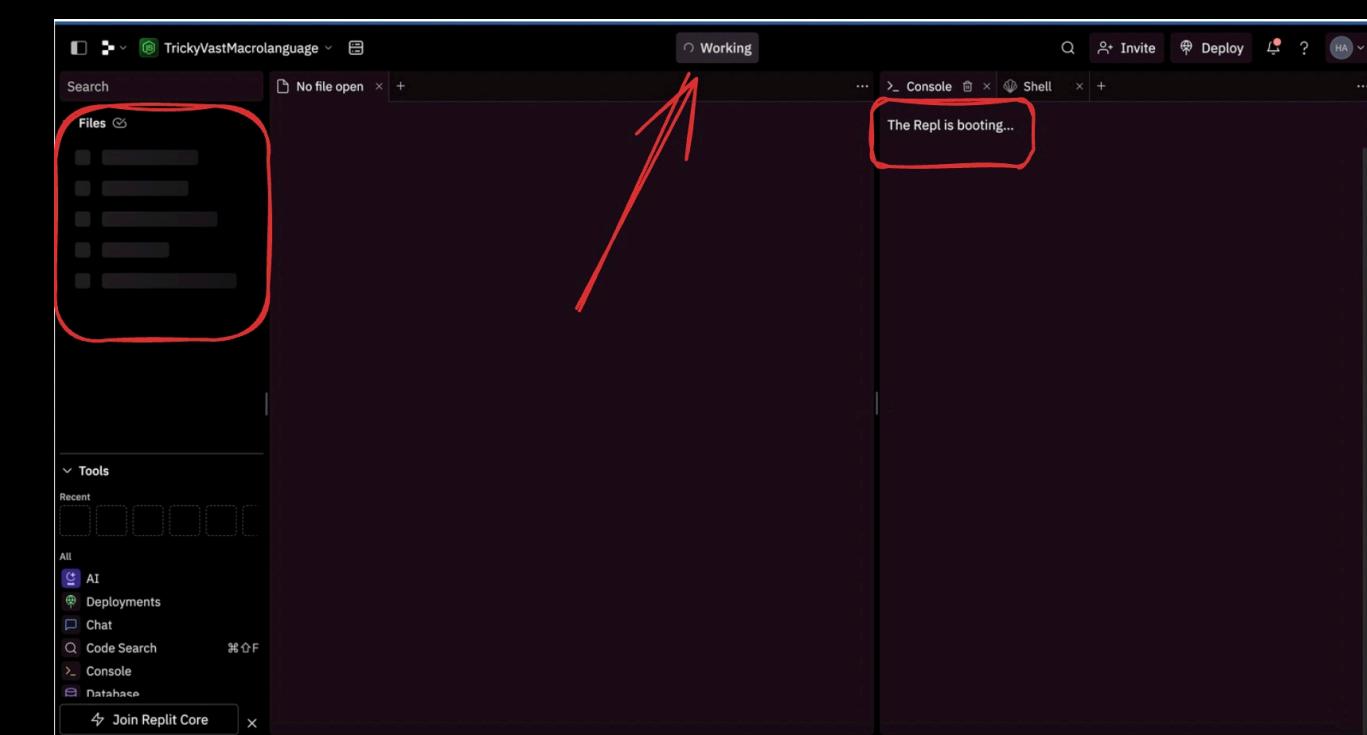


http

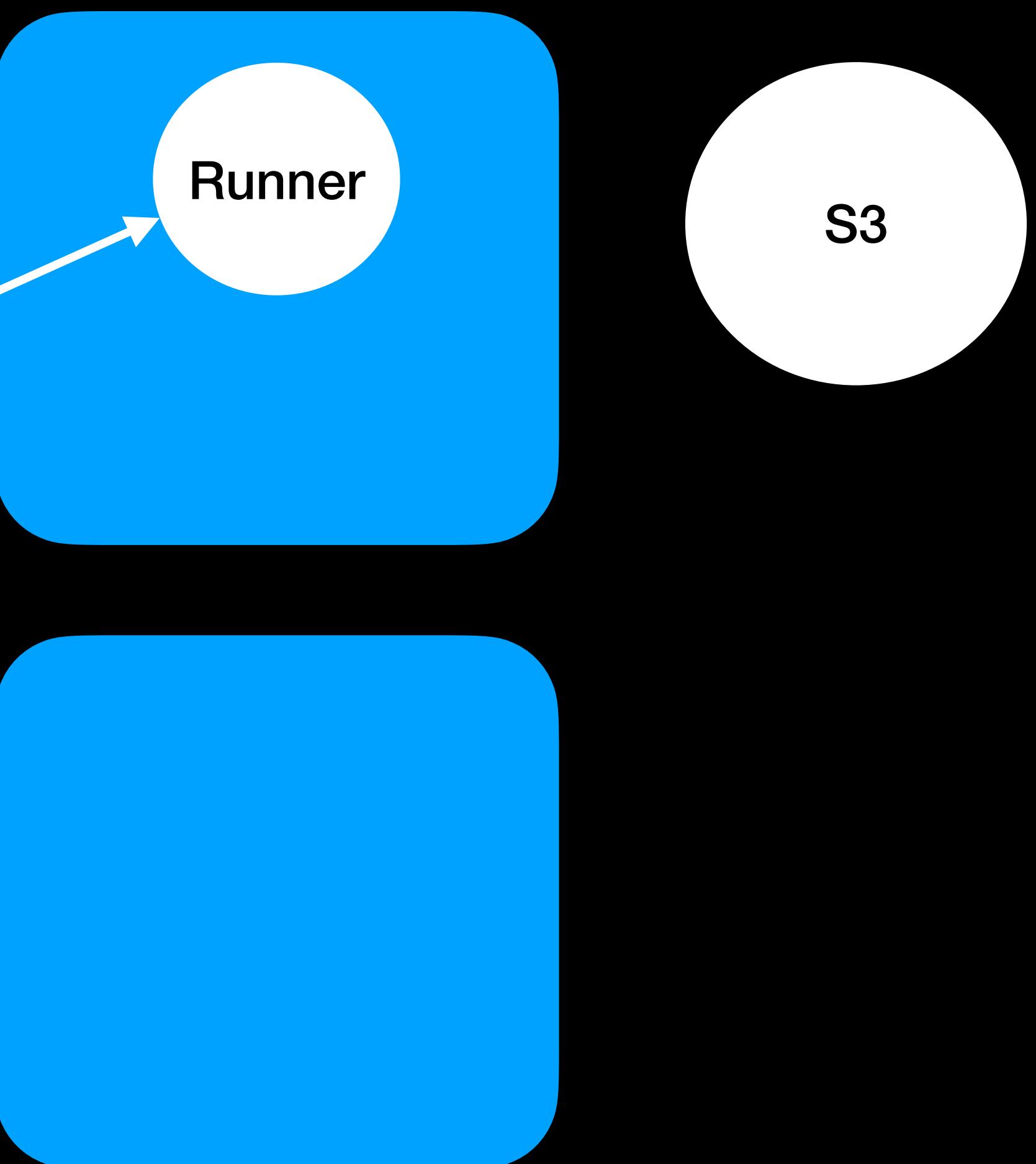
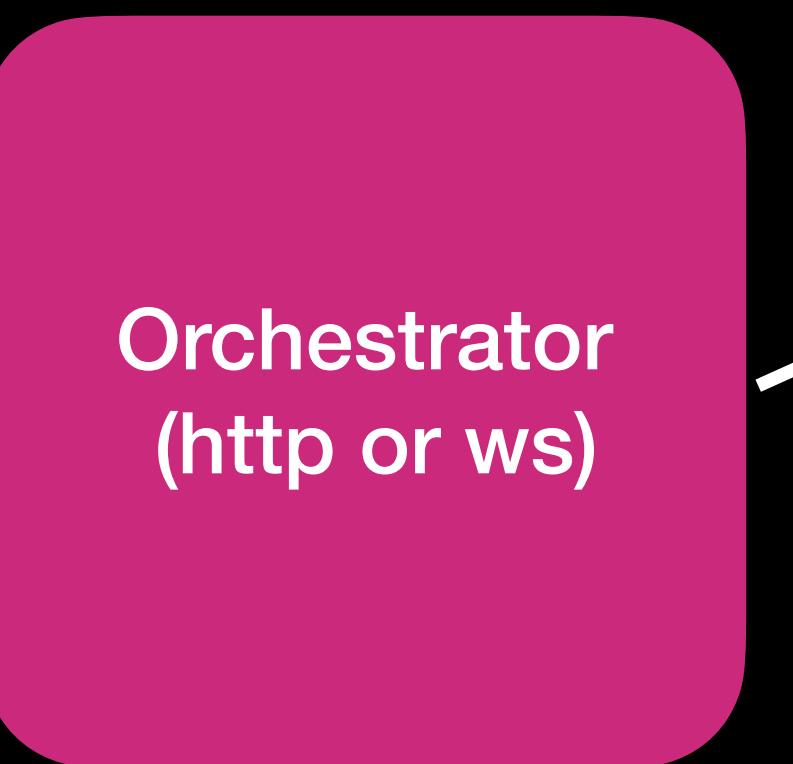


Orchestrator

Step 3 -Tell the orchestrator to start a pod

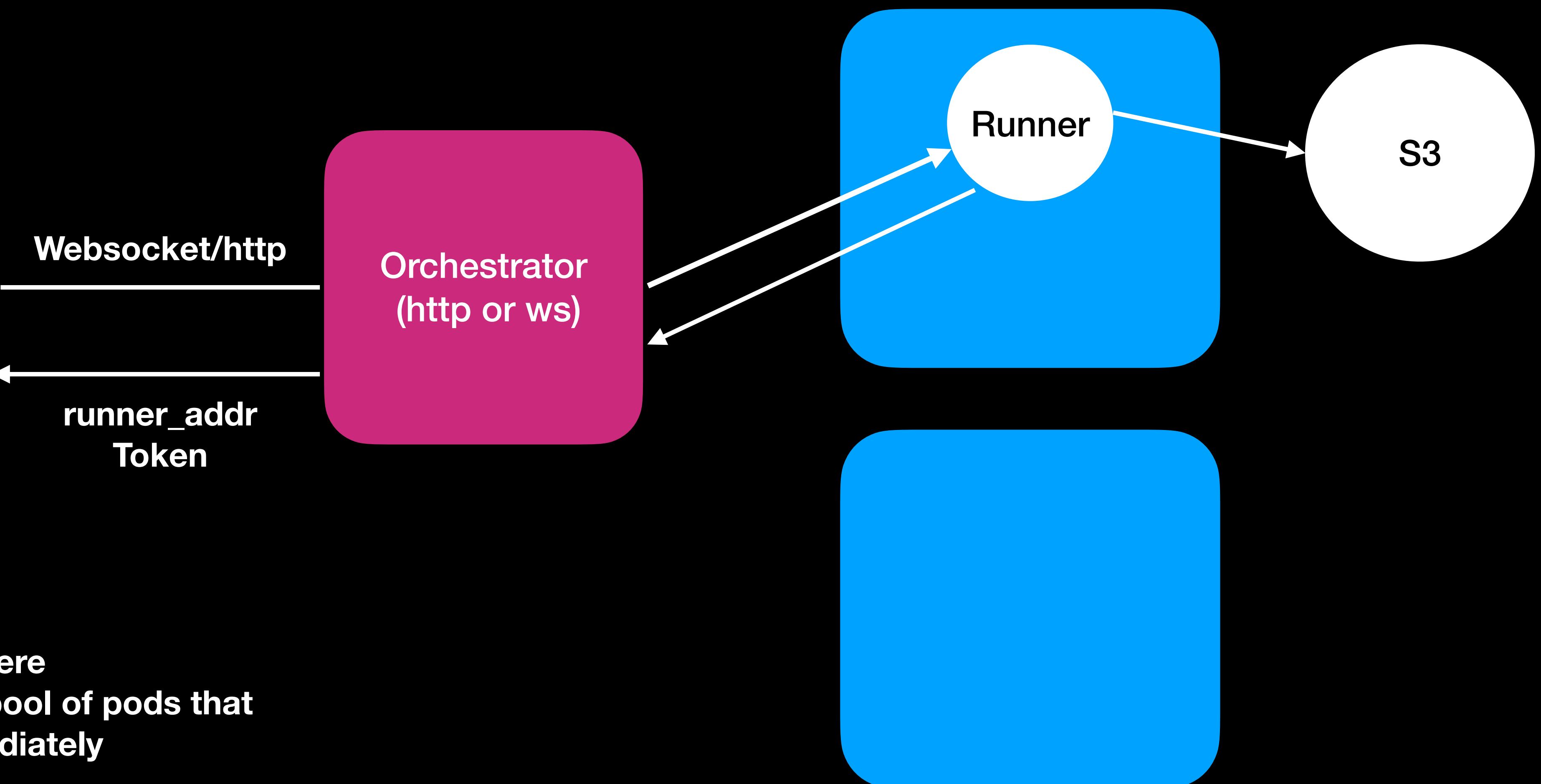
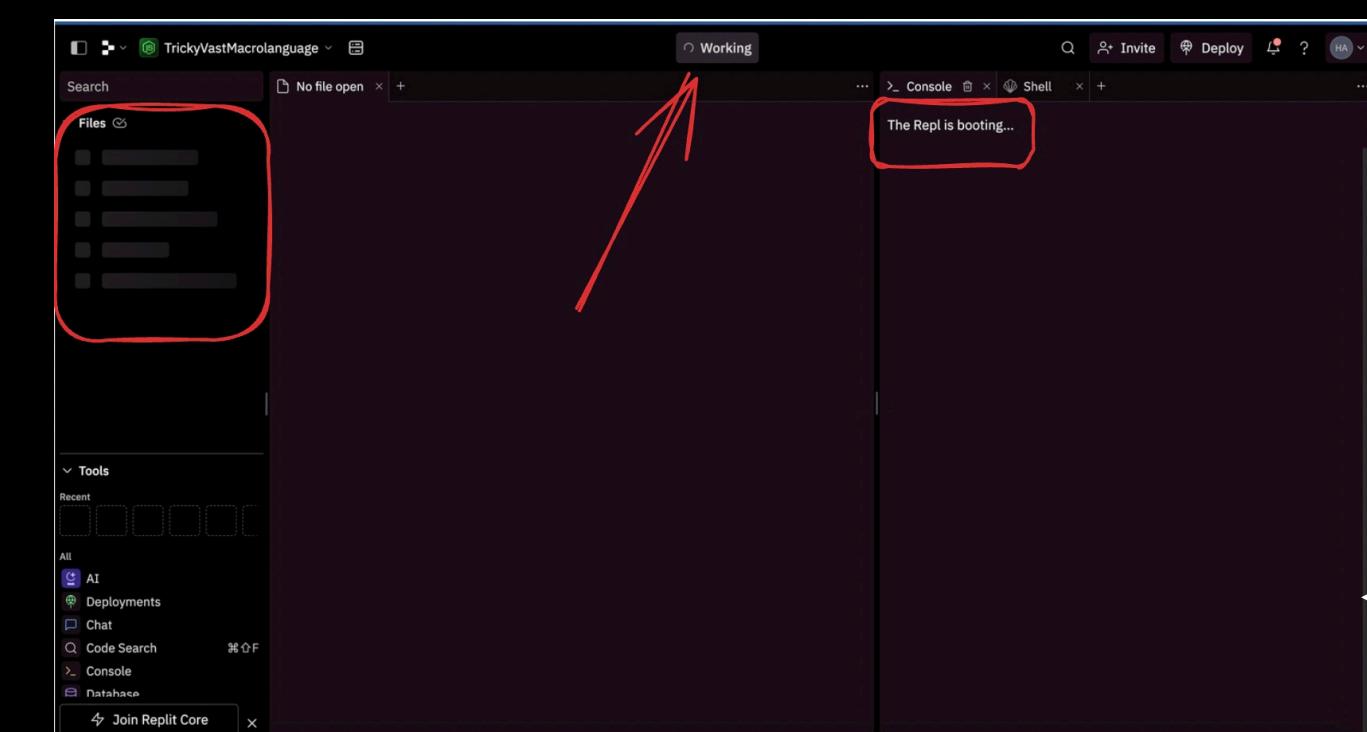


Websocket/http



Execution service

Step 3 -Tell the orchestrator to start a pod
Tells it to pull the code from S3

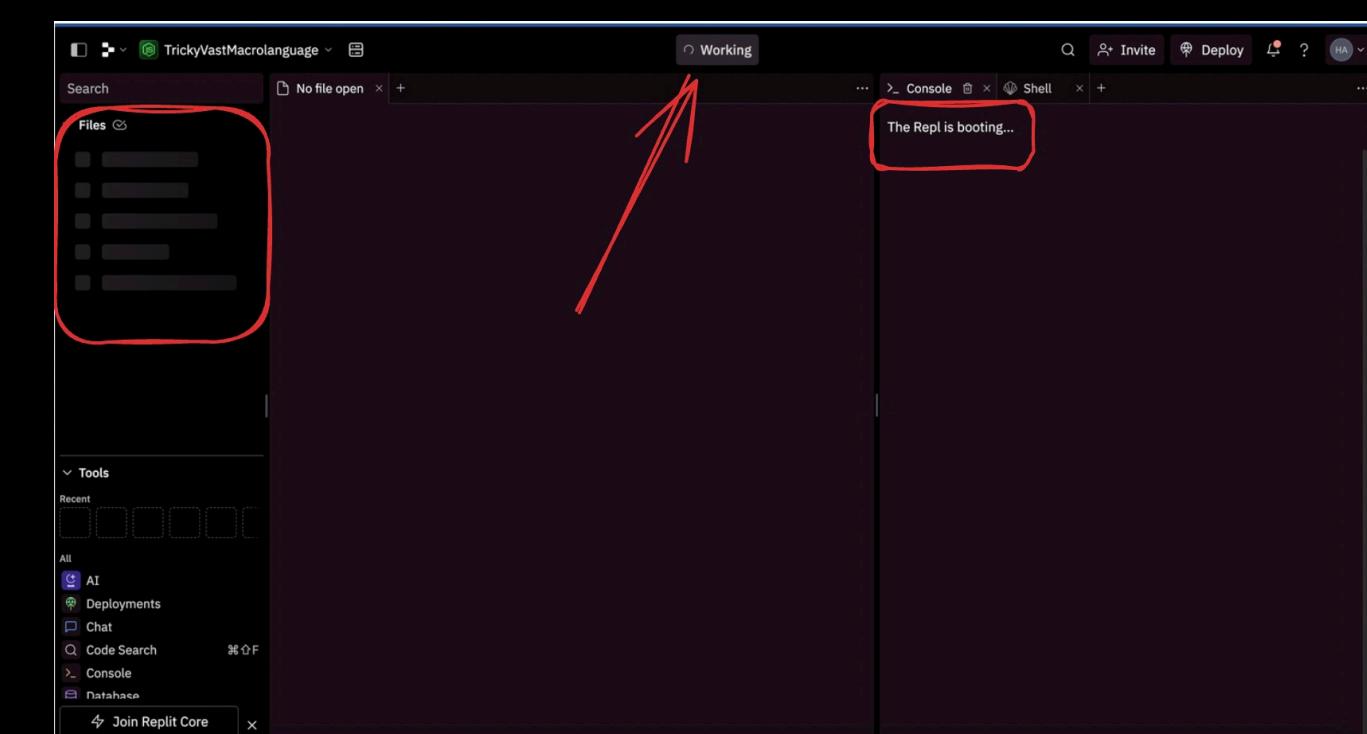


Callout -

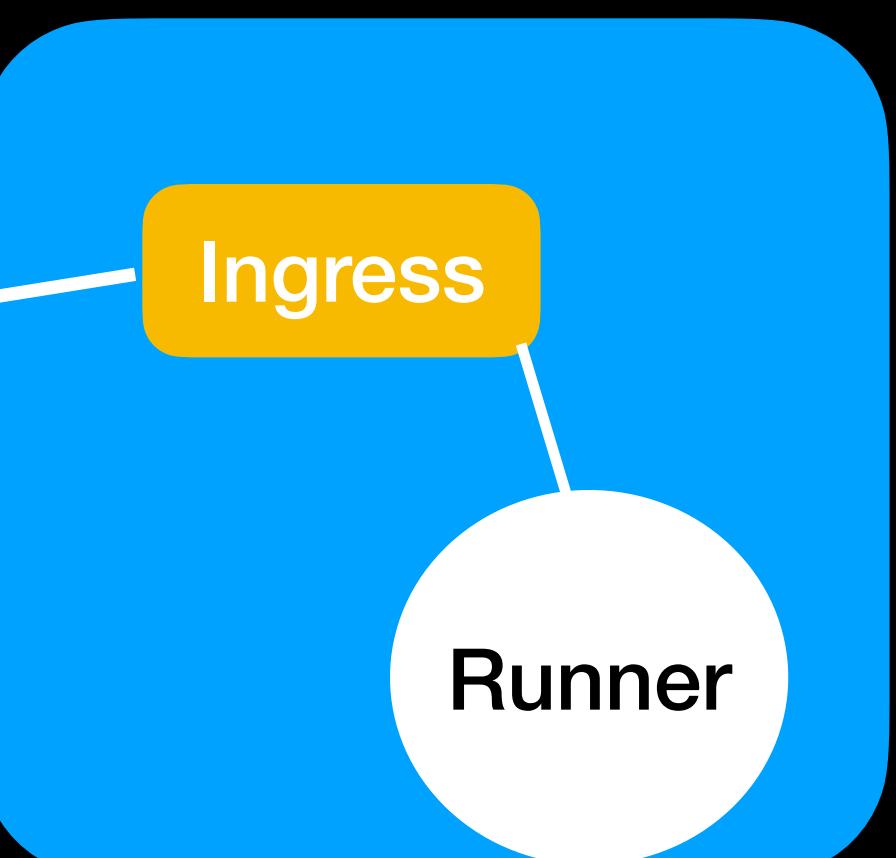
1. Caching is super helpful here
2. You can maintain a warm pool of pods that you can auto assign immediately

Execution service

Step 3 -Tell the orchestrator to start a pod



runner_addr
Token
Websocket

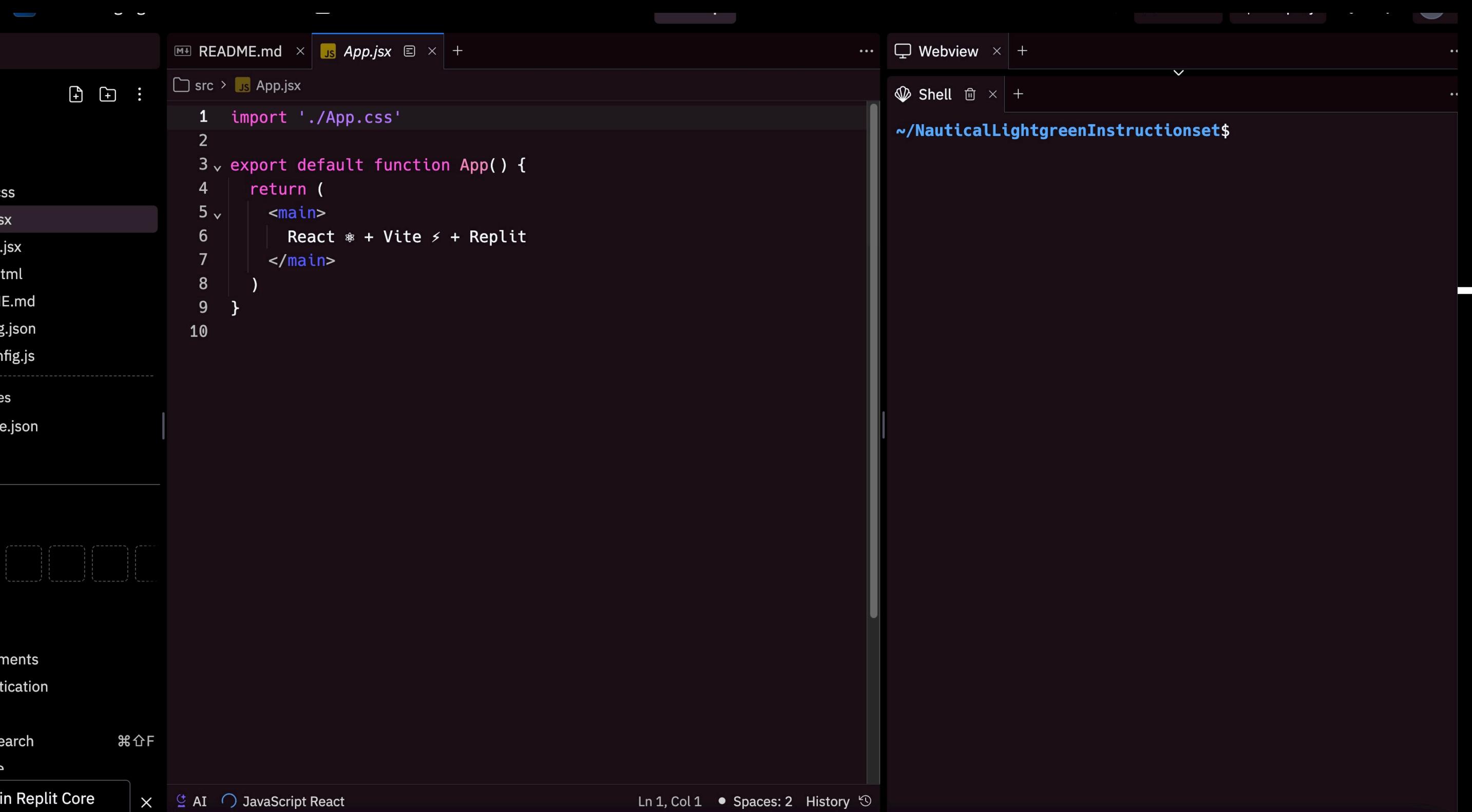


Callout -

1. Caching is super helpful here
2. You can maintain a warm pool of pods that you can auto assign immediately

Execution service

Step 4 - Let the user edit a file, send over diff over the ws layer



A screenshot of a dark-themed code editor. The left pane shows a file tree with files like README.md, App.css, App.jsx, and others. The main pane displays the contents of App.jsx:

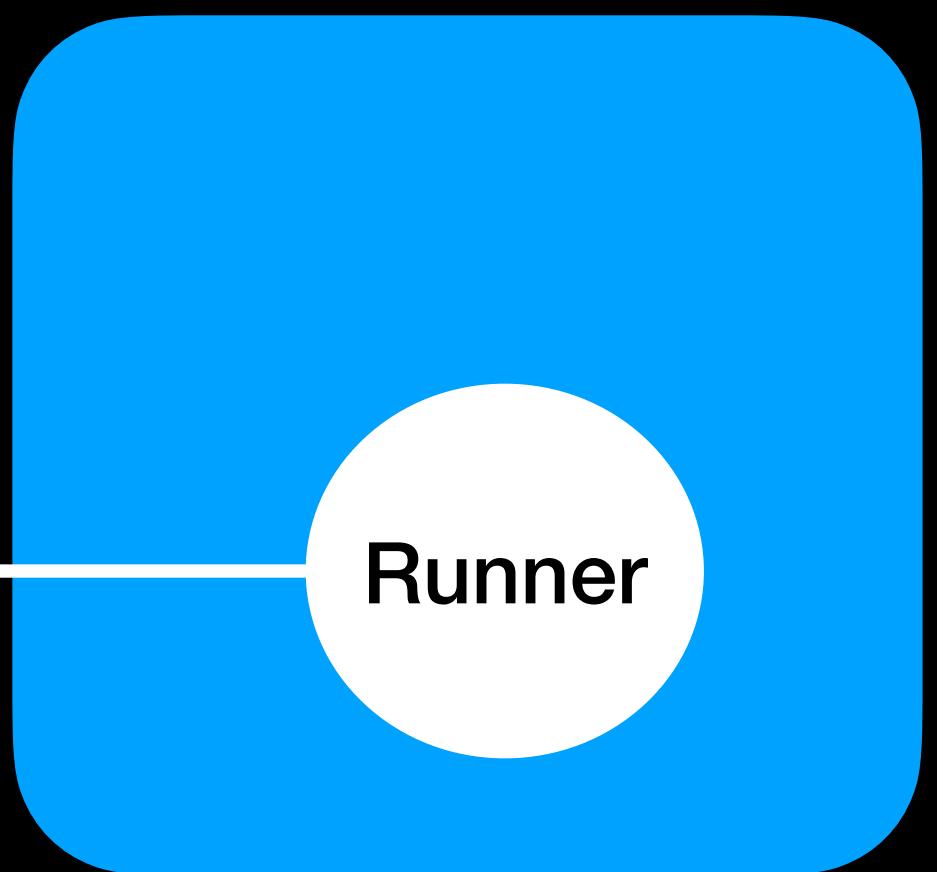
```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       | React * + Vite ↴ + Replit  
7     </main>  
8   )  
9 }  
10
```

The right pane contains a "Shell" tab with the text: ~/NauticalLightgreenInstructionset\$.

runner_addr
Token

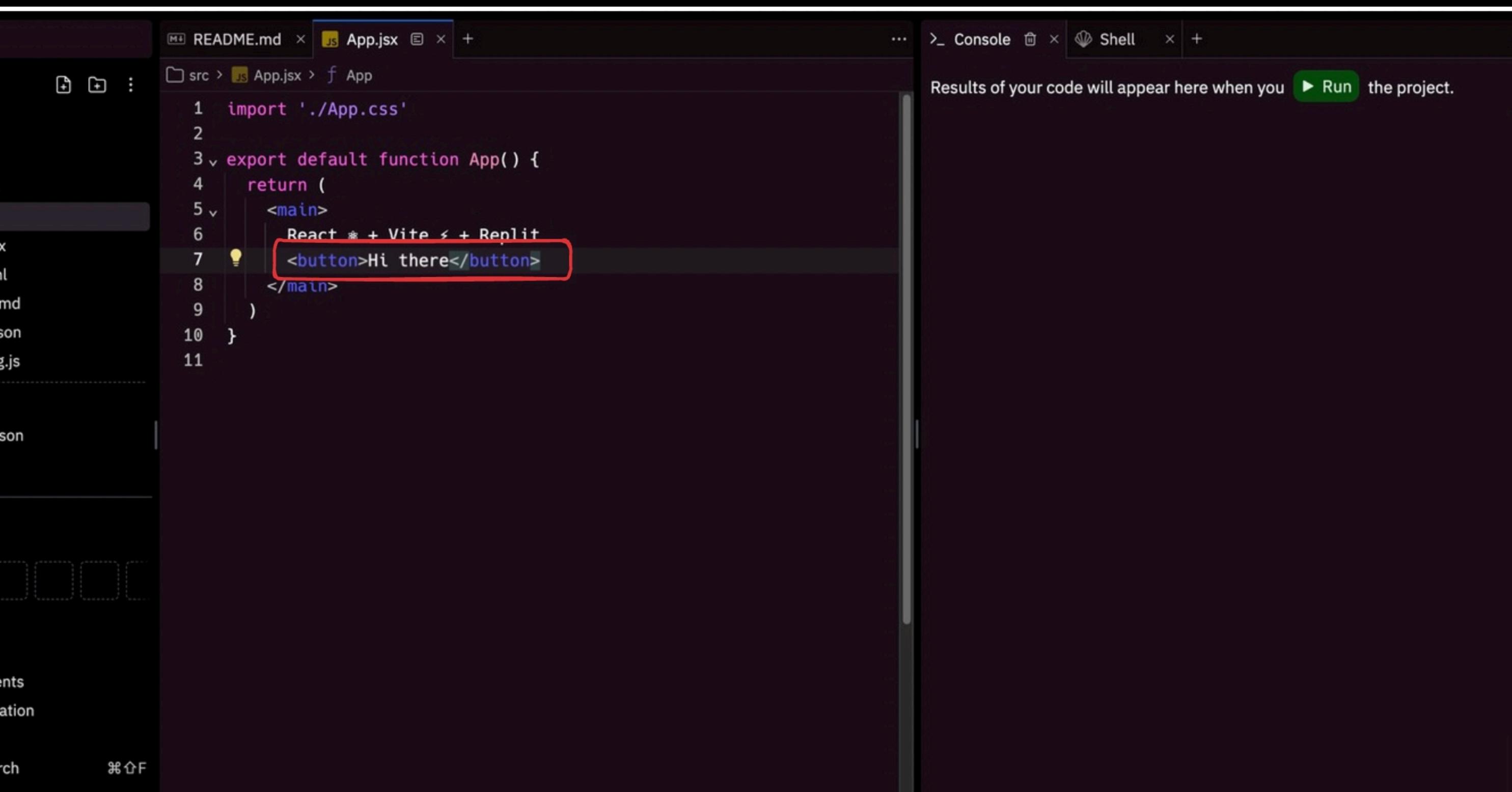
websocket

Runner



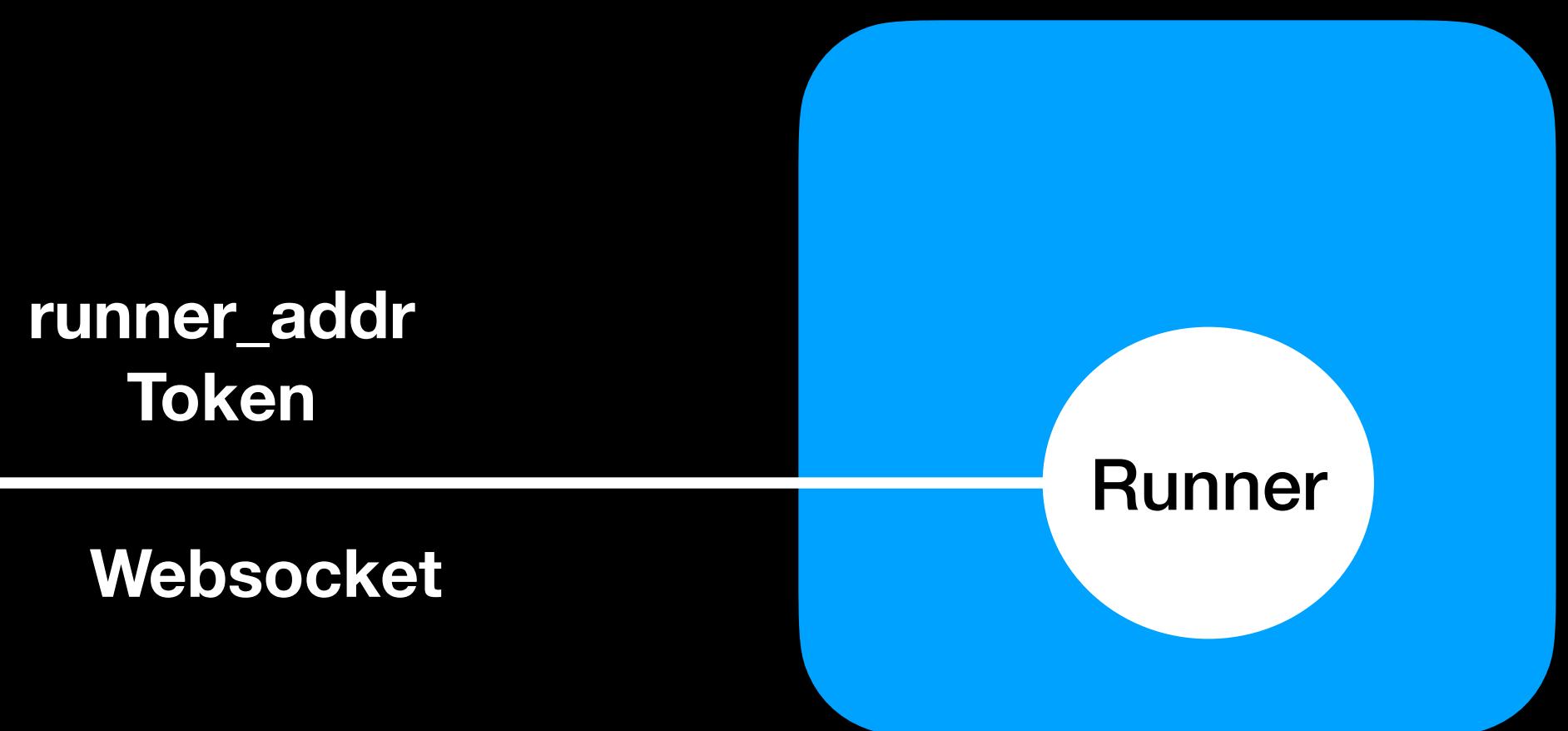
Execution service

Step 4 - Let the user edit a file, send over diff over the ws layer



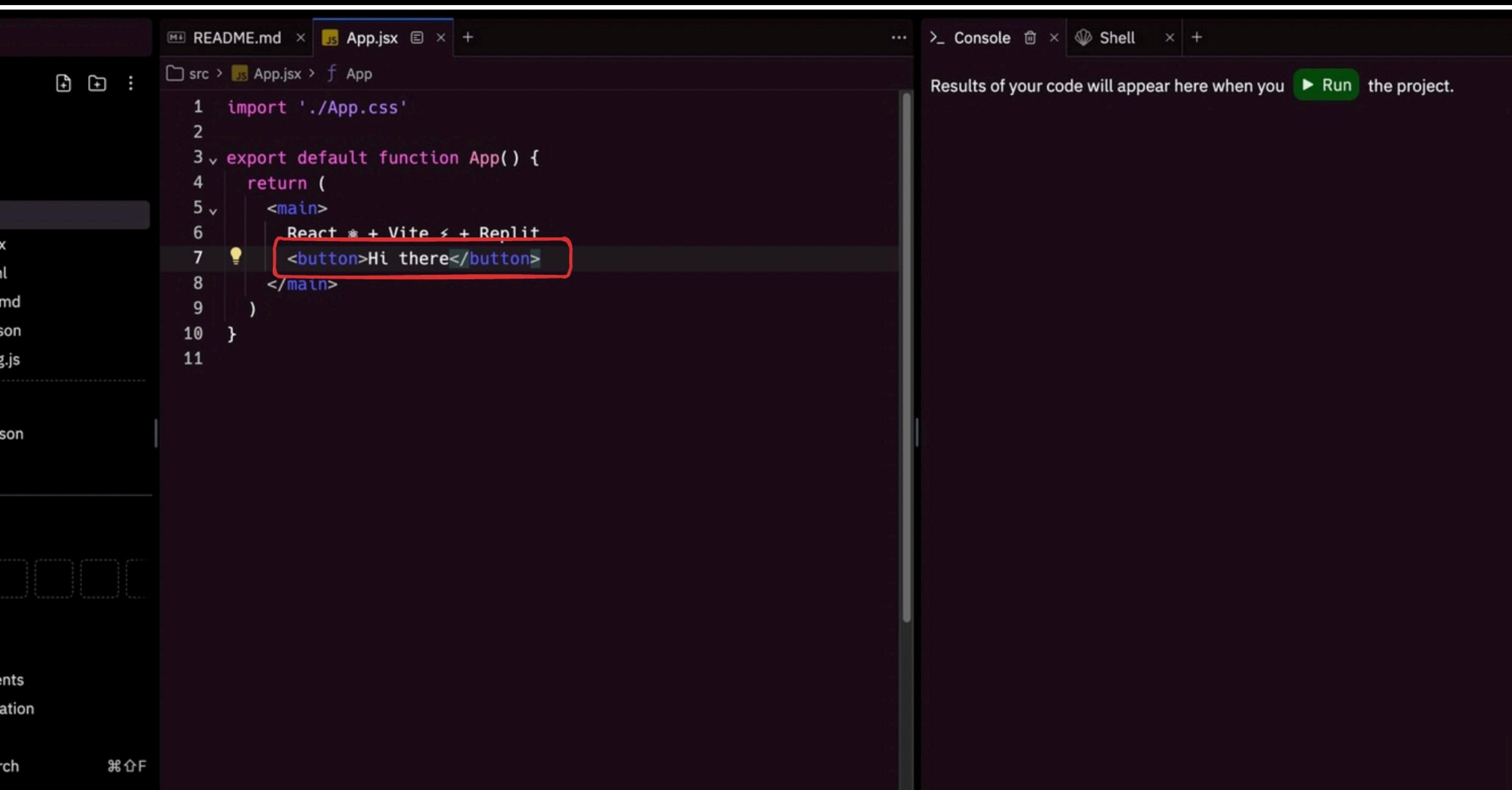
A screenshot of a code editor showing an `App.jsx` file. The code contains a `<button>Hi there</button>` component. A red box highlights this line of code. The editor interface includes tabs for `README.md`, `App.jsx`, `Console`, and `Shell`. A message in the `Console` tab says, "Results of your code will appear here when you Run the project."

```
1 import './App.css'  
2  
3 export default function App() {  
4   return (  
5     <main>  
6       React * + Vite < + Replit  
7       <button>Hi there</button>  
8     </main>  
9   )  
10 }  
11
```



Execution service

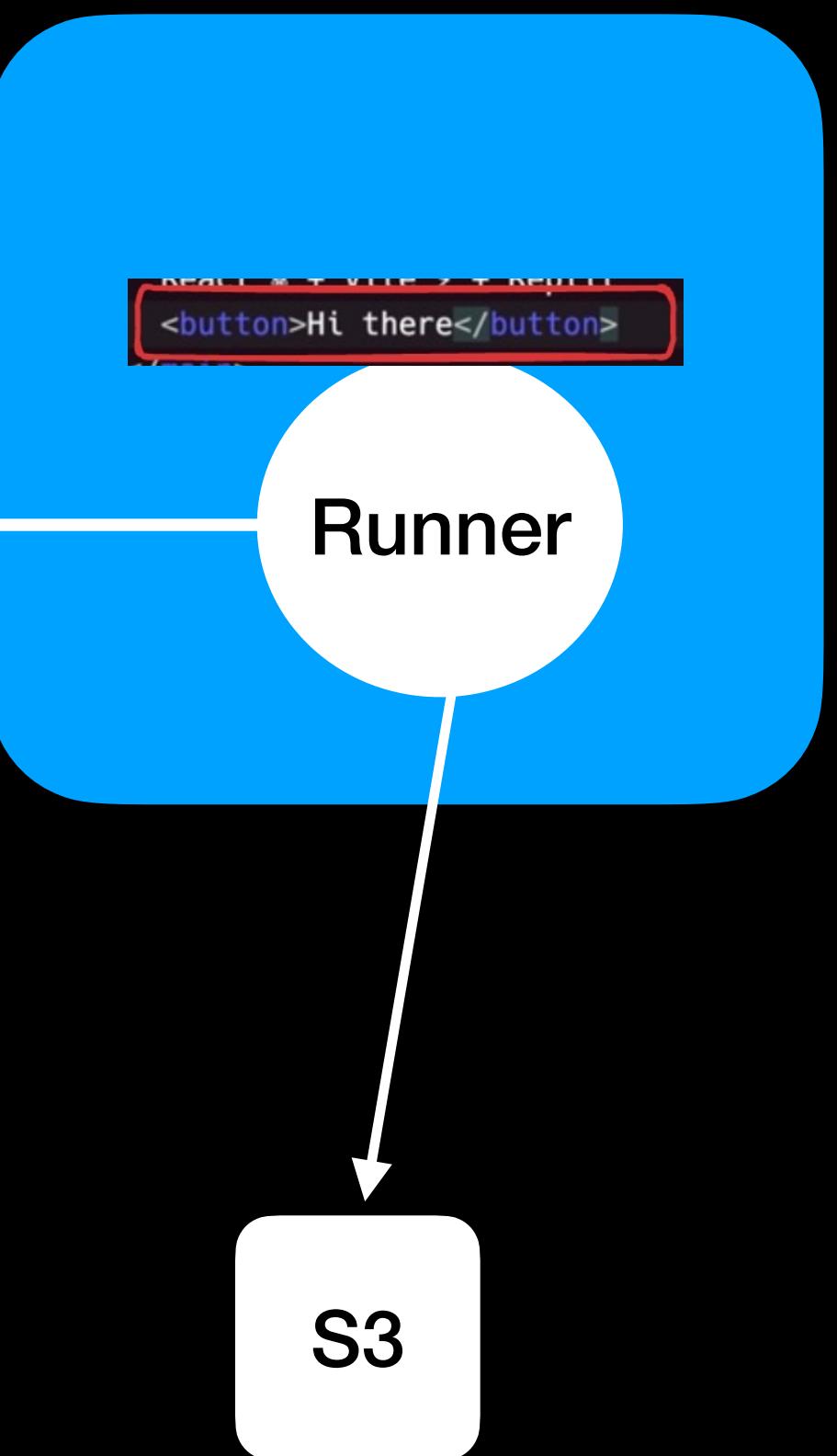
Step 4 - Let the user edit a file, send over diff over the ws layer



A screenshot of a code editor showing a file named App.jsx. The code contains a simple React component:import './App.css'
export default function App() {
 return (
 <main>
 React * + Vite < + Replit
 <button>Hi there</button>
 </main>
)
}
The line containing the button component is highlighted with a red box.

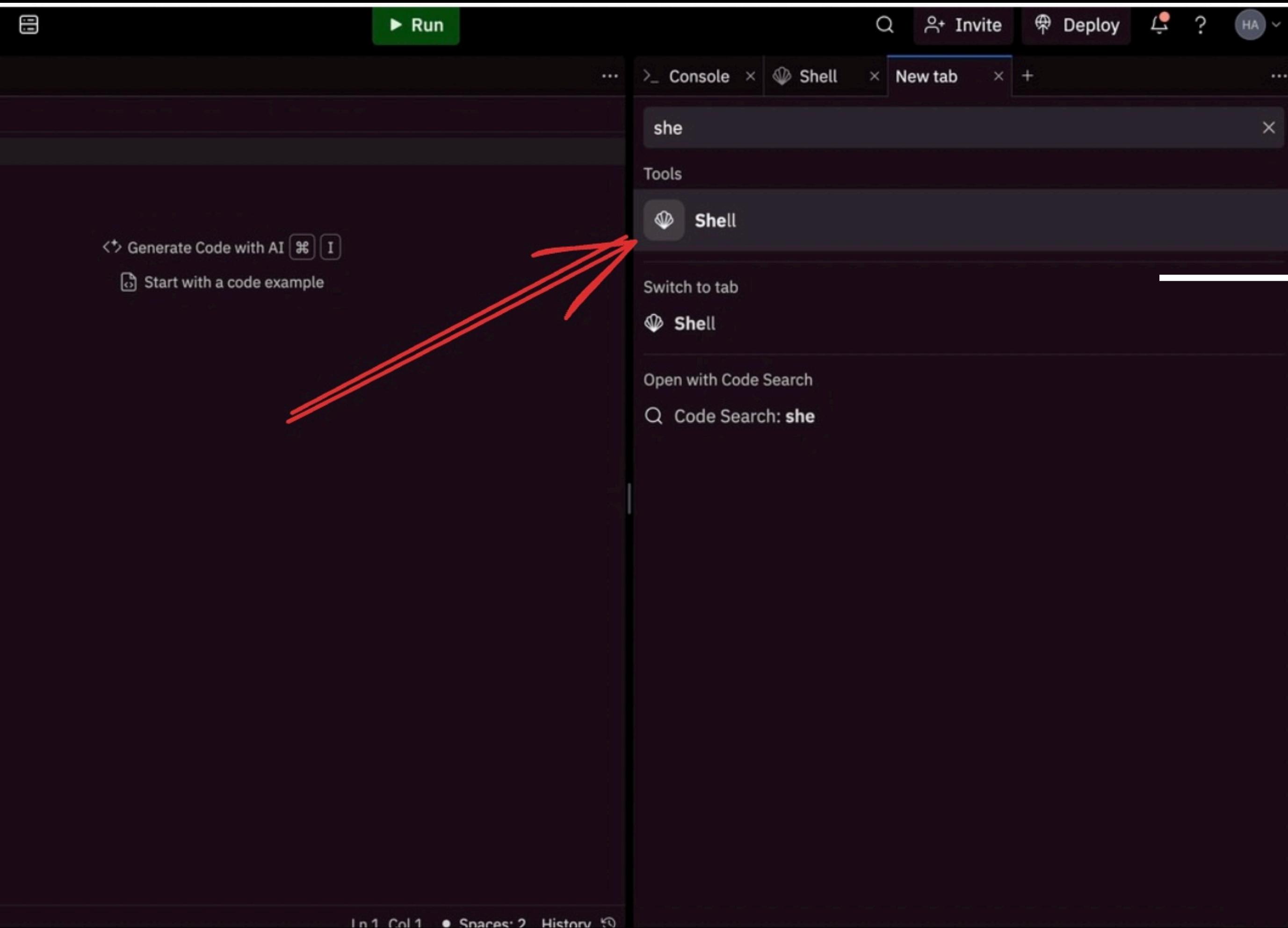
runner_addr
Token

Websocket



Execution service

Step 5 - Terminal access



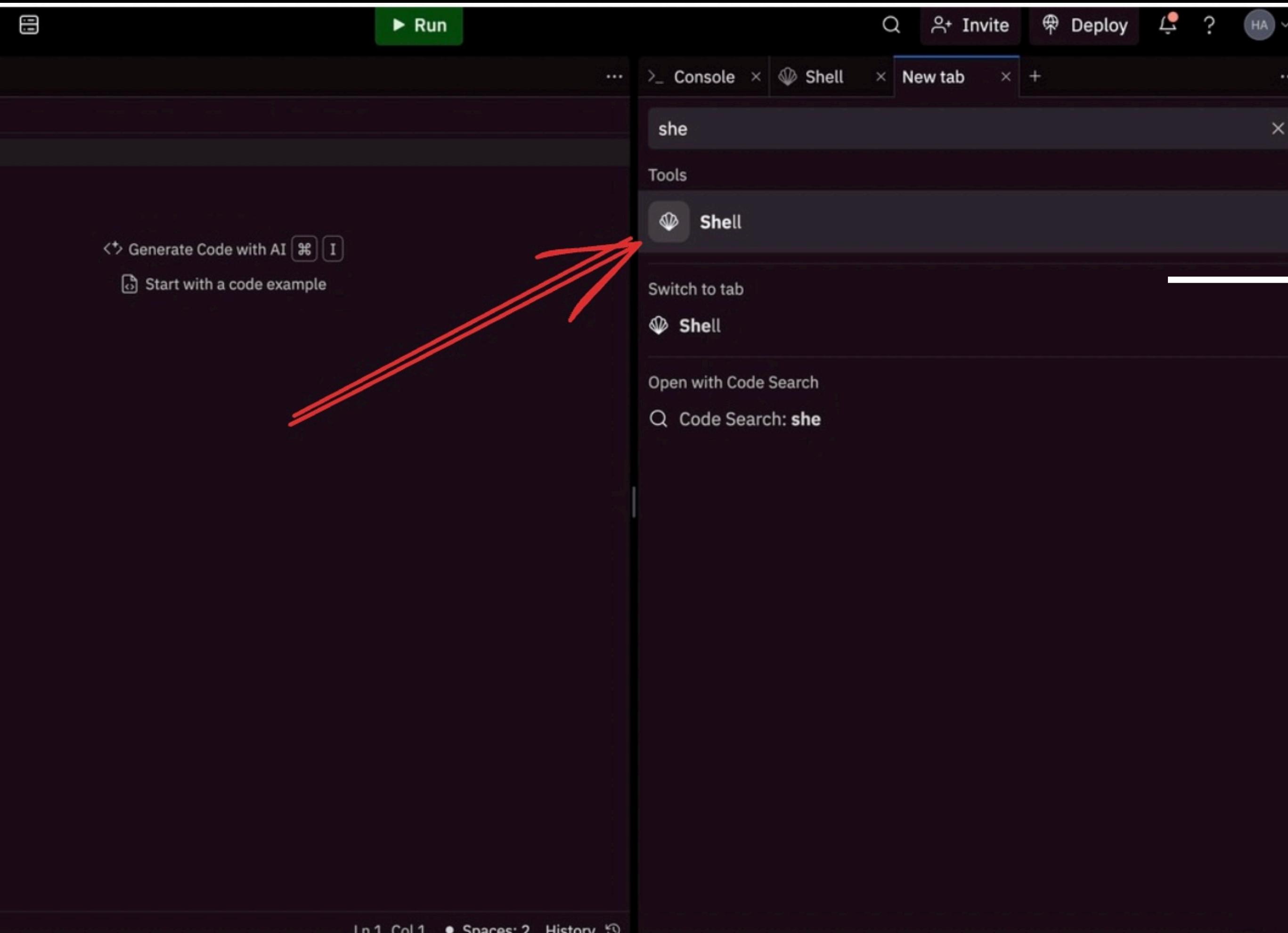
startSession

Websocket

Runner

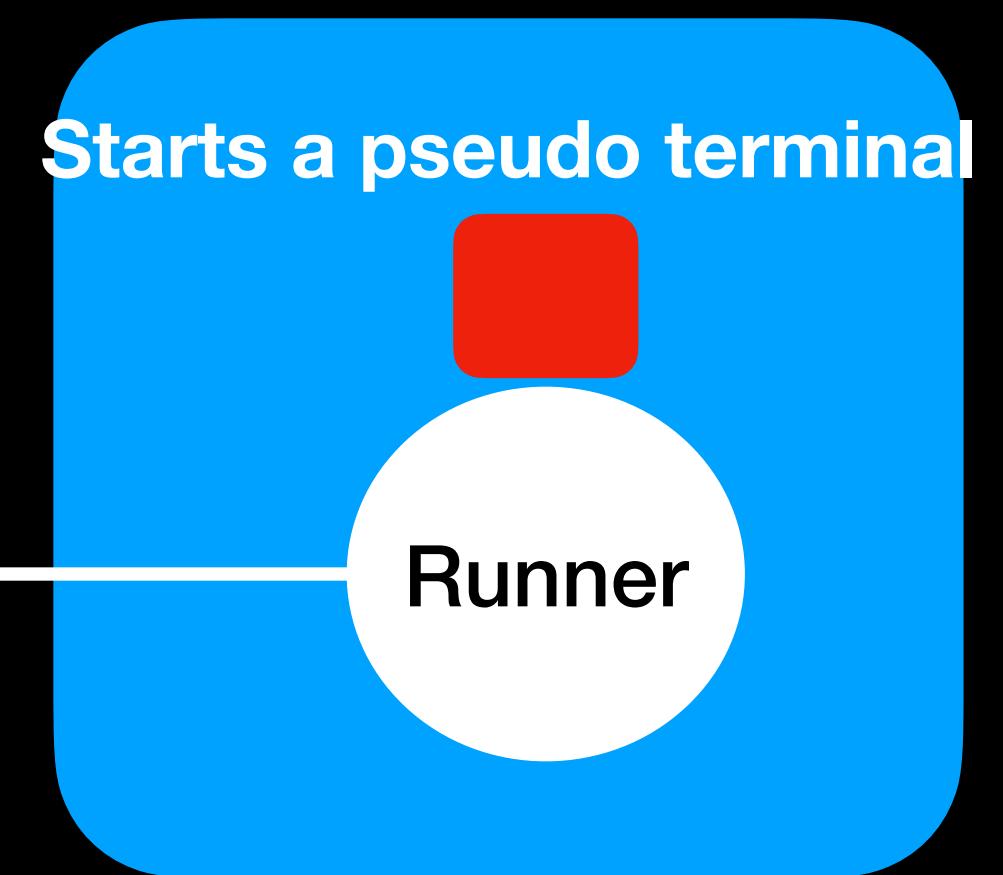
Execution service

Step 5 - Terminal access

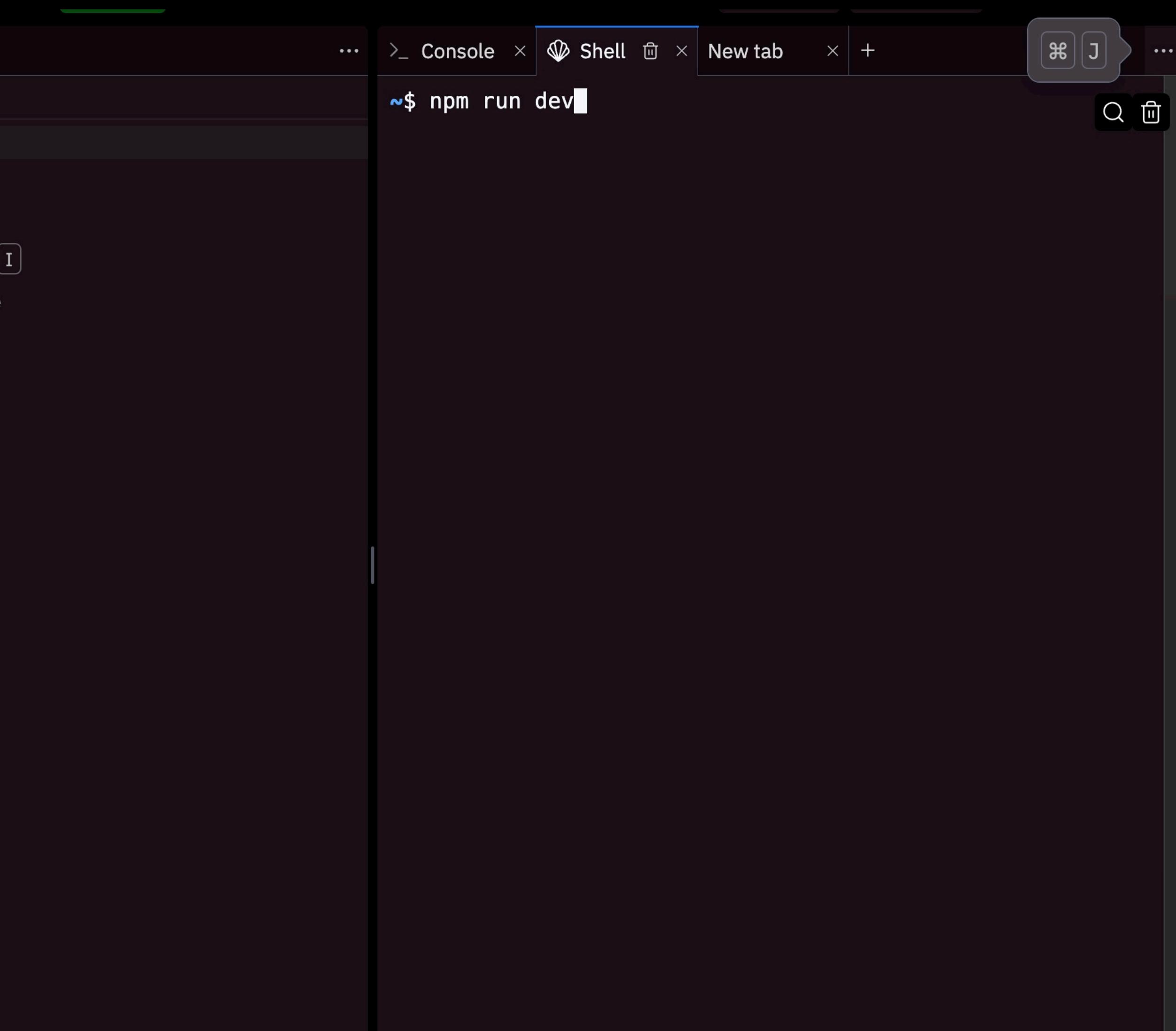


startSession

Websocket



Execution service



A screenshot of a terminal window with a dark theme. The tab bar shows three tabs: 'Console', 'Shell', and 'New tab'. The 'Shell' tab is active, displaying the command `~$ npm run dev`. The terminal interface includes standard controls like a search bar and a trash bin icon.

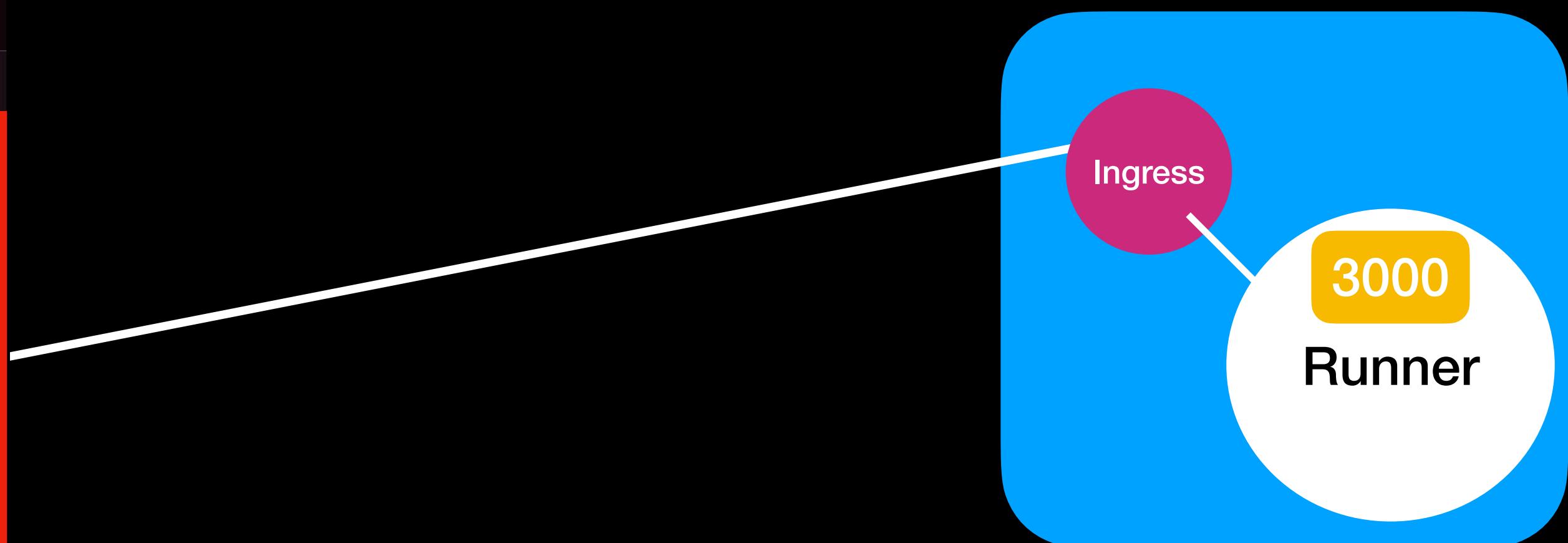
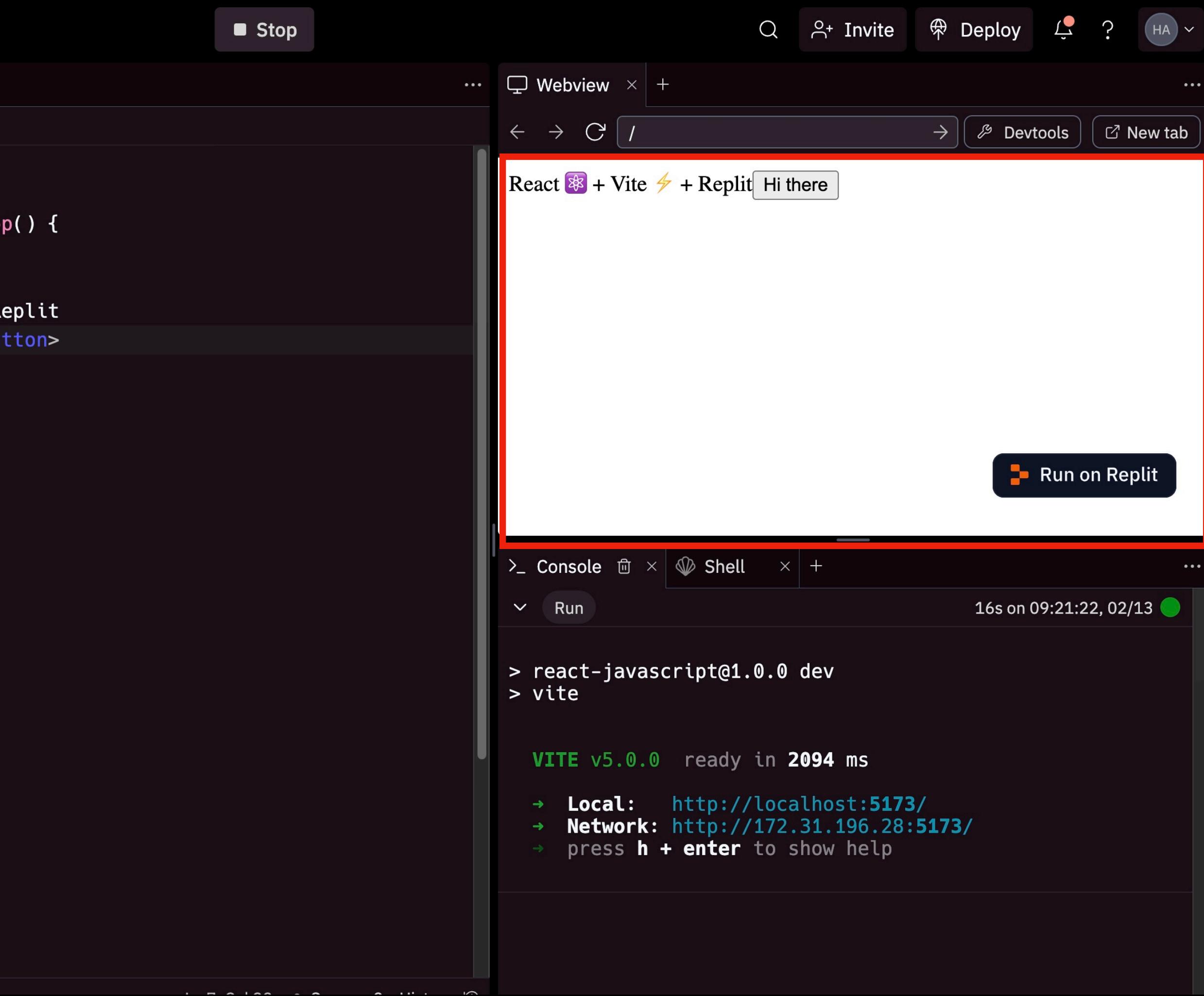
Step 5 - Terminal access

Relay keystrokes/commands



Execution service

Step 6 - Accessing a process



Execution service

Step 7 - Destroying the pod

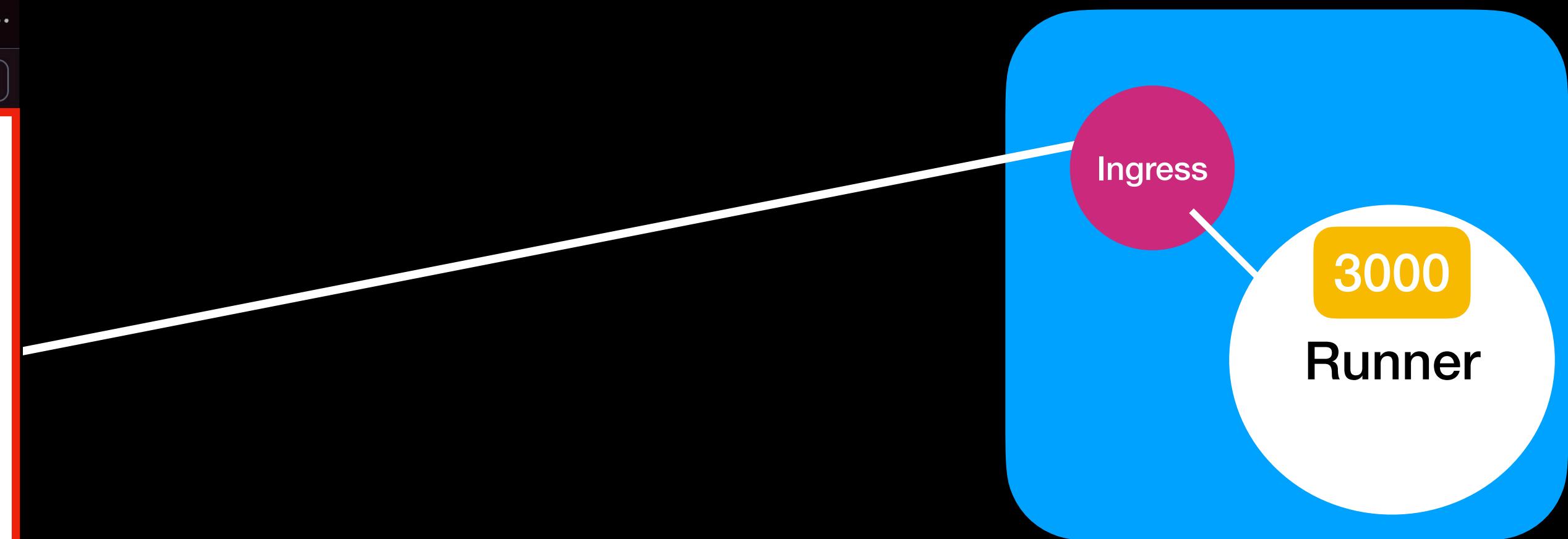
If the process has 0 ws conns alive for
~5 minutes, it can kill itself.

A screenshot of the Replit IDE interface. At the top, there's a navigation bar with a 'Stop' button, a search icon, an 'Invite' button, a 'Deploy' button, a help icon, and a 'HA' dropdown. Below the navigation bar is a browser window titled 'Webview' showing a React application with the message 'React + Vite + Replit Hi there'. A red box highlights this browser window. At the bottom of the browser window is a 'Run on Replit' button. Below the browser window is a terminal window with tabs for 'Console' and 'Shell'. The 'Console' tab is active and shows the command output:

```
> react-javascript@1.0.0 dev
> vite

VITE v5.0.0  ready in 2094 ms

→ Local:  http://localhost:5173/
→ Network: http://172.31.196.28:5173/
→ press h + enter to show help
```



Execution service

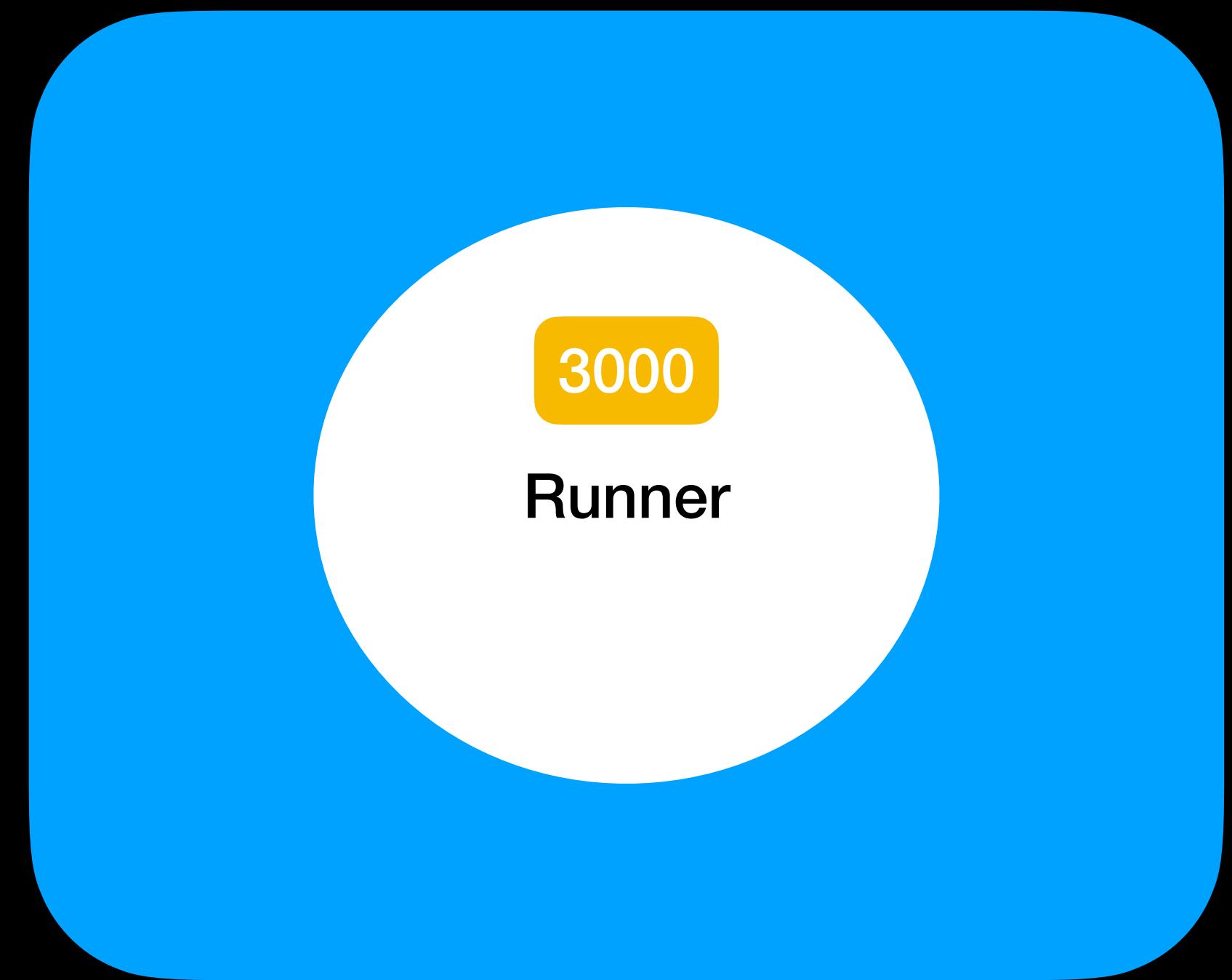
Why is this secure?

Remote code runs in a pod

Pod has restricted permissions

Pod has resource limits (only 2CPUs for example)

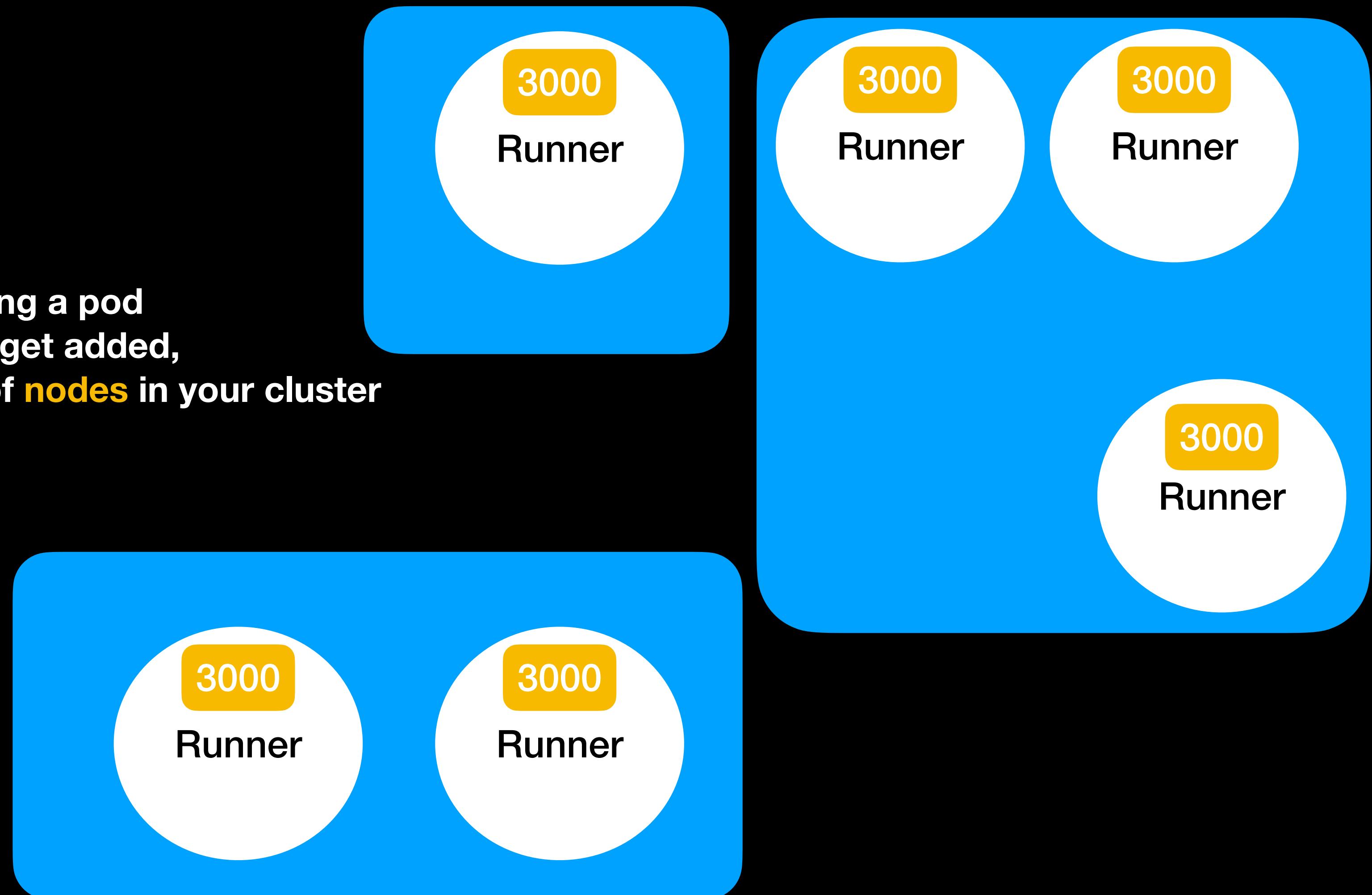
Pod has its own network (no port conflict)



Execution service

Why is this scalable?

Starting a new runner is as simple as starting a pod
As the CPU usage goes up and more pods get added,
All you have to do is increase the number of nodes in your cluster



Few callouts that might feel like good to haves

- 1. User can inspect your ws codebase (not good). They can stop this process as well.**
Fixing this might involve starting a parent process which takes inputs from the user and forwards it to the container through a socket
- 2. User can still reboot your machine (can do the same on replit)**

Few advance things that repl.it does

1. **Nix for reproducible builds**
2. **Network mounts to allow users to get 250Gb space on each repl**
3. **repl.it doesn't let the user directly connect to the pod, but most probably relays information via a relay ws server**