Rspack

The fast Rust-based web bundler

Rspack

- Introduction & Comparison
- Migration & Demo
- Rust

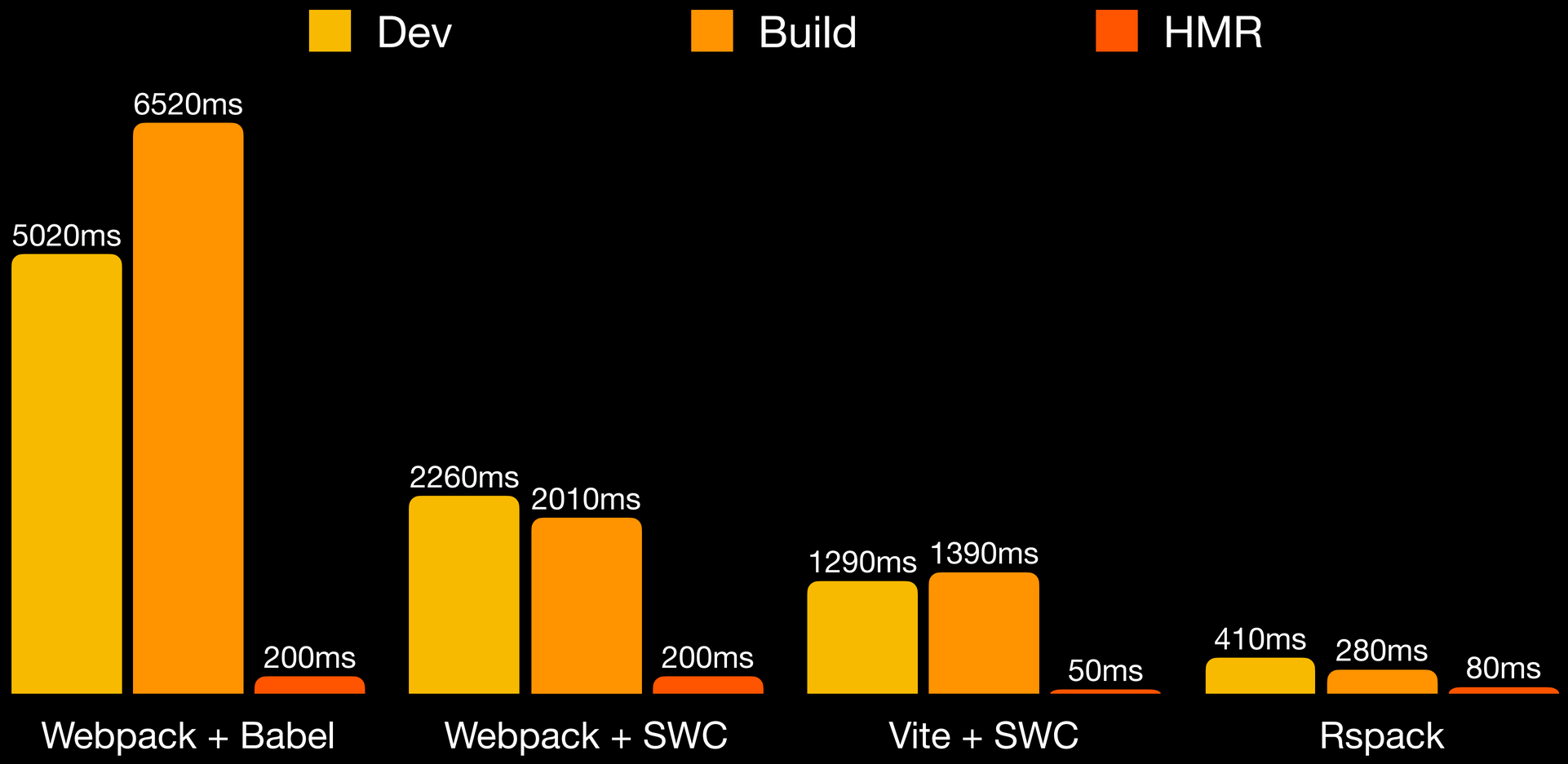
What’s Rspack?

- High performance JavaScript bundler written in Rust
- Created by ByteDance Web Infra team
- Rspack 1.0 is released in August 2024

Why Rspack?

- Fast dev mode startup, HMR and builds
- Webpack compatible
- Community ecosystem for modern web development

Lightning Fast



Who uses Rspack

Microsoft

amazon

ByteDance

TikTok

Alibaba

Discord

Who uses Rspack

Docusaurus

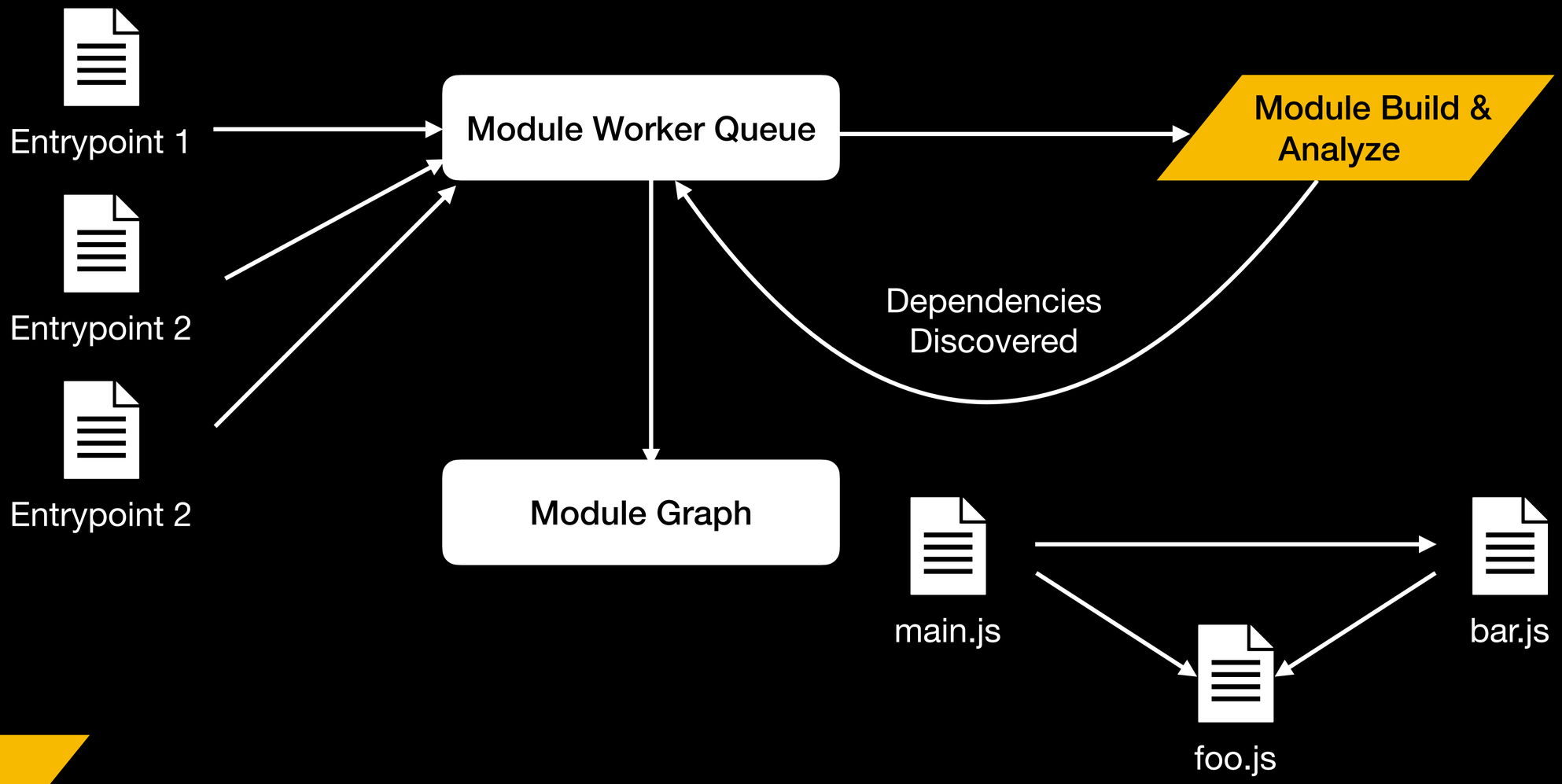
Nuxt

Storybook

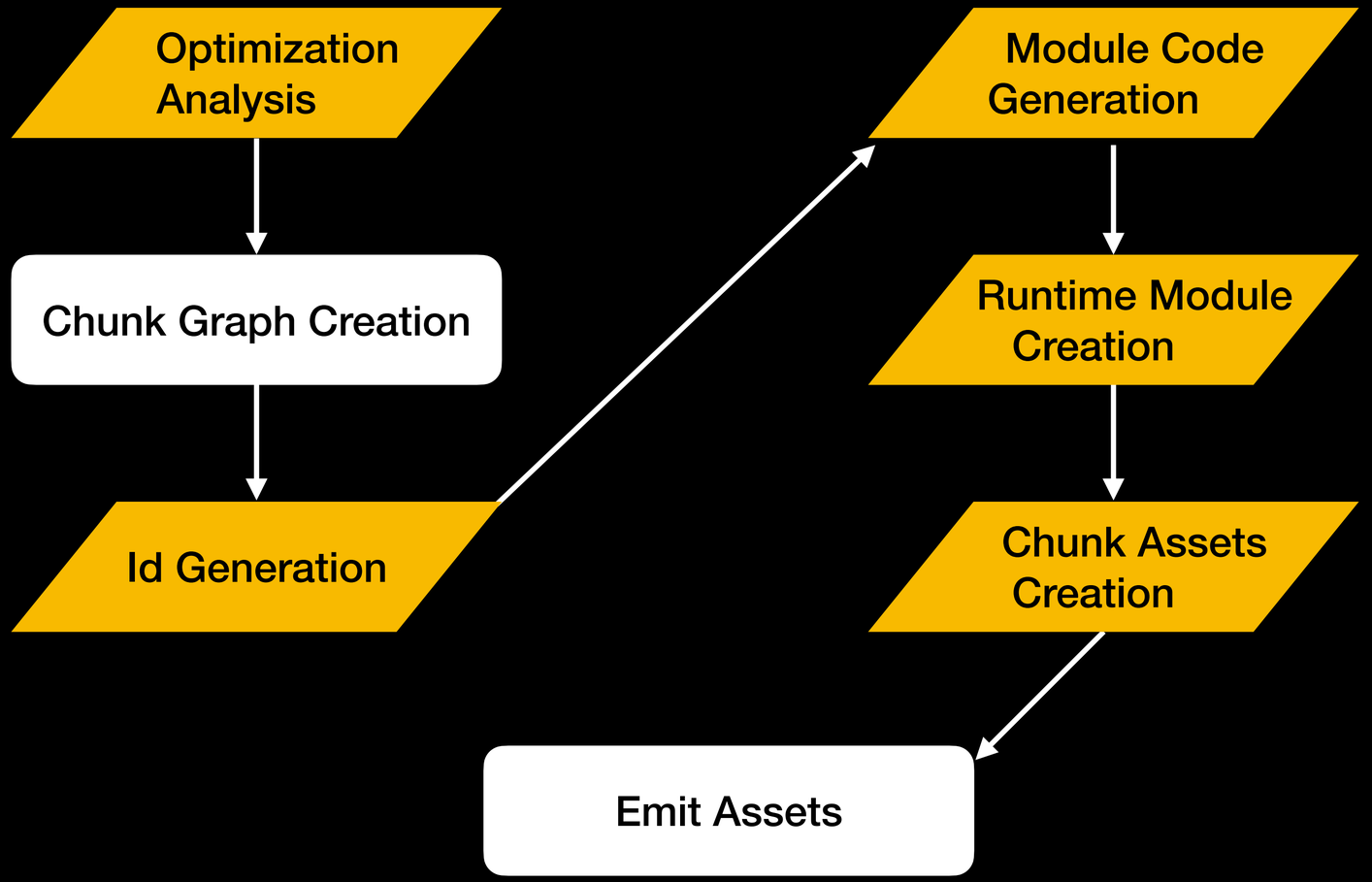
Compared with Webpack

- Rust language efficiency
    - Speed
    - Memory management
    - Highly parallelized architecture
- Loaders and plugins are also efficiency

Make phase in Rspack



Seal phase in Rspack



Compared with Vite

- No bundling in development mode
    - Browser handles import and export with Vite
- Reliance on Rollup for production builds
    - Rust version Rollup: Rolldown

Comparison

	Current Version	Rust Version
Compiler	 Babel	 SWC
Bundler	 Webpack	 Rspack

Vite Comparison

	Current Version	Rust Version
Compiler	 Esbuild	 Oxc
Bundler	 Rollup	 Rolldown

It’s Rust Age

Lightning Fast JavaScript Tooling

Migrate from Webpack to Rspack

- Base on webpack 5
- Install Rspack
- Replace all plugins and loaders with Rspack version
    - SWC Loader target

Install Rspack

- pnpm add @rspack/core @rspack/cli --save-dev
- pnpm remove webpack webpack-cli webpack-dev-server

Updating package.json

```
{
  "scripts": {
    "serve": "webpack serve",
    "build": "webpack build"
  }
}
```

Updating package.json

```
{
  "scripts": {
    "serve": "rspack serve",
    "build": "rspack build"
  }
}
```

Update plugins

```
const CopyWebpackPlugin = require('copy-webpack-plugin');

module.exports = {
  plugins: [
    new CopyWebpackPlugin({
      // ...
    }),
  ]
}
```

Update plugins

```
const rspack = require('@rspack/core');

module.exports = {
  plugins: [
    new rspack.CopyRspackPlugin({
      // ...
    }),
  ]
}
```

Update loaders

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/i,
        use: ["file-loader"],
      },
    ],
  },
};
```

Update loaders

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpe?g|gif)$/i,
        type: "asset/resource",
      },
    ],
  },
};
```

Migrate from existing projects

- Migrating from Webpack to Rspack
- Migrating from Create React App to Rsbuild
- NestJS
- And more

Community Ecosystem

Rsbuild

Rslib

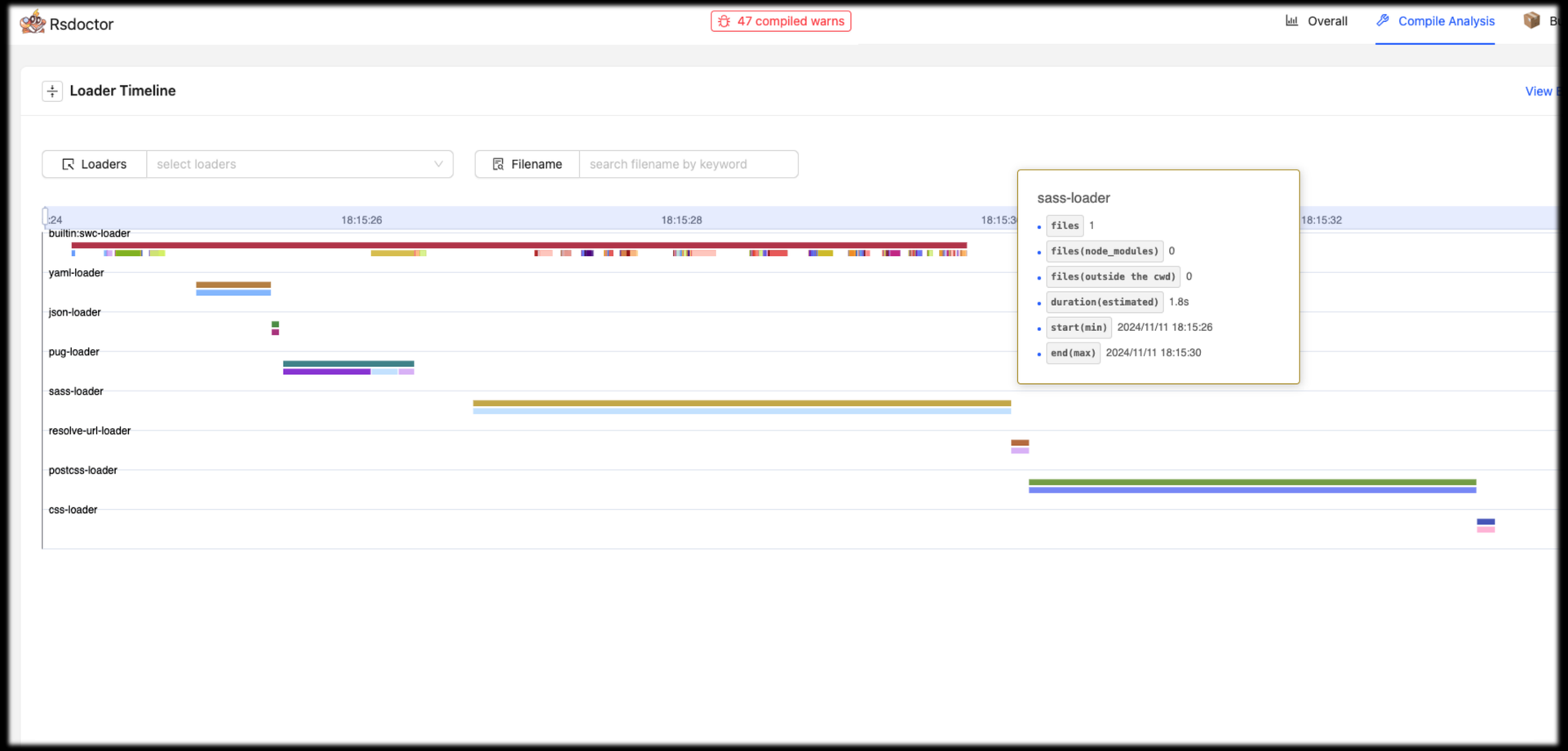
Rspress

Rsdoctor

Awesome
Rspack

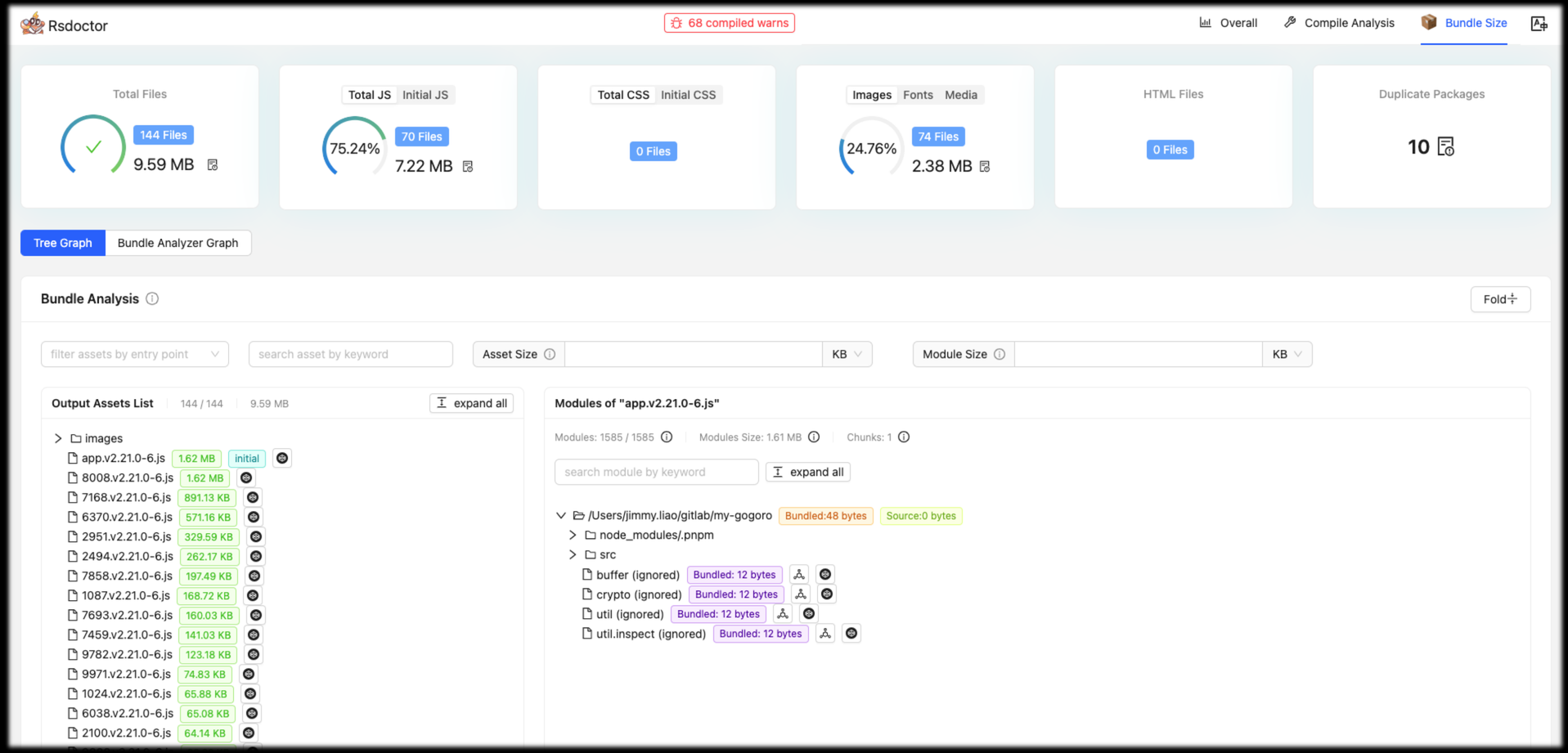
Rsdoctor

- Visualize the building process
    - Compile’s Time
    - Bundle Analysis



Rsdoctor

Loader Timeline



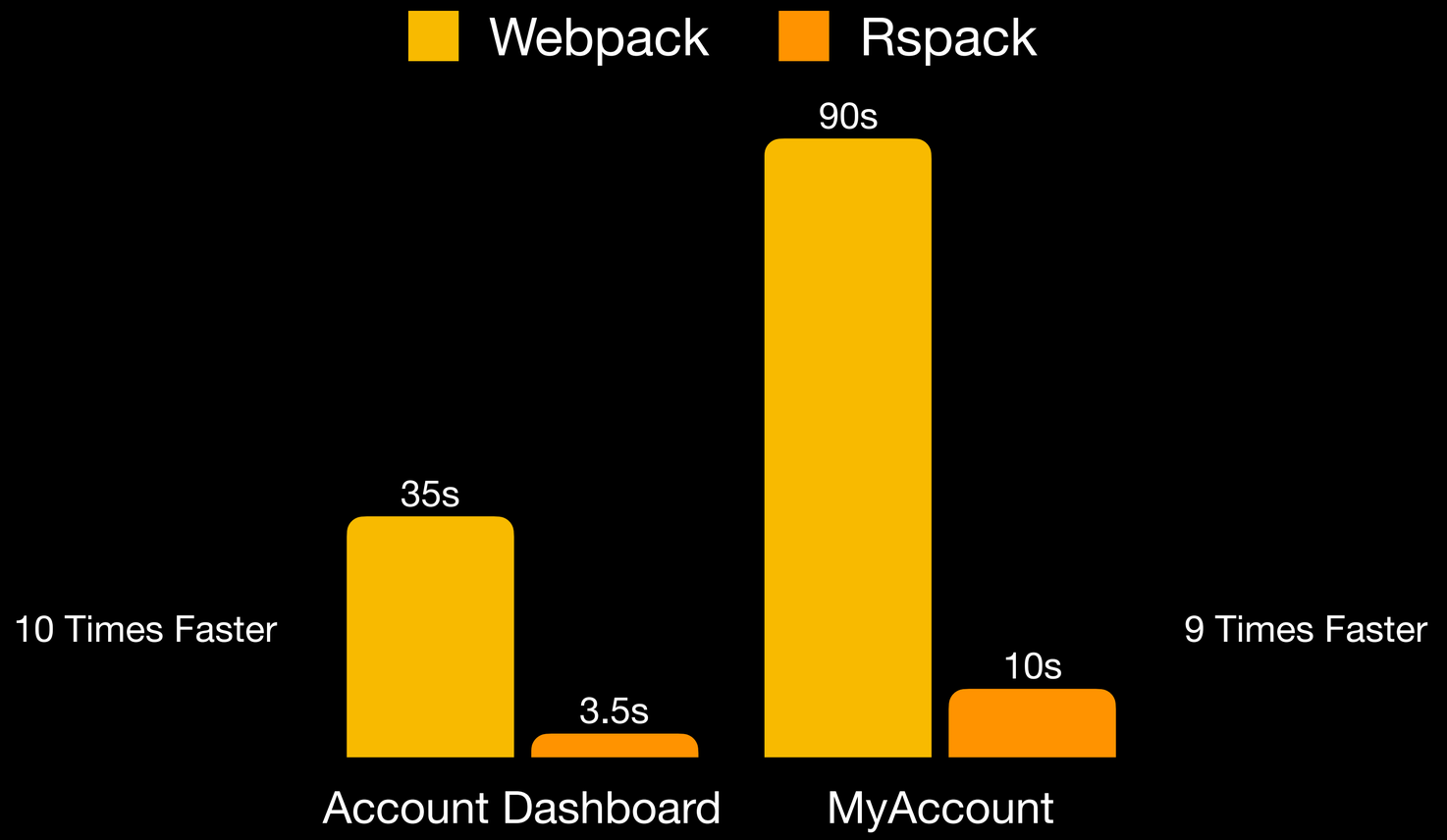
Rsdoctor

Bundle Size

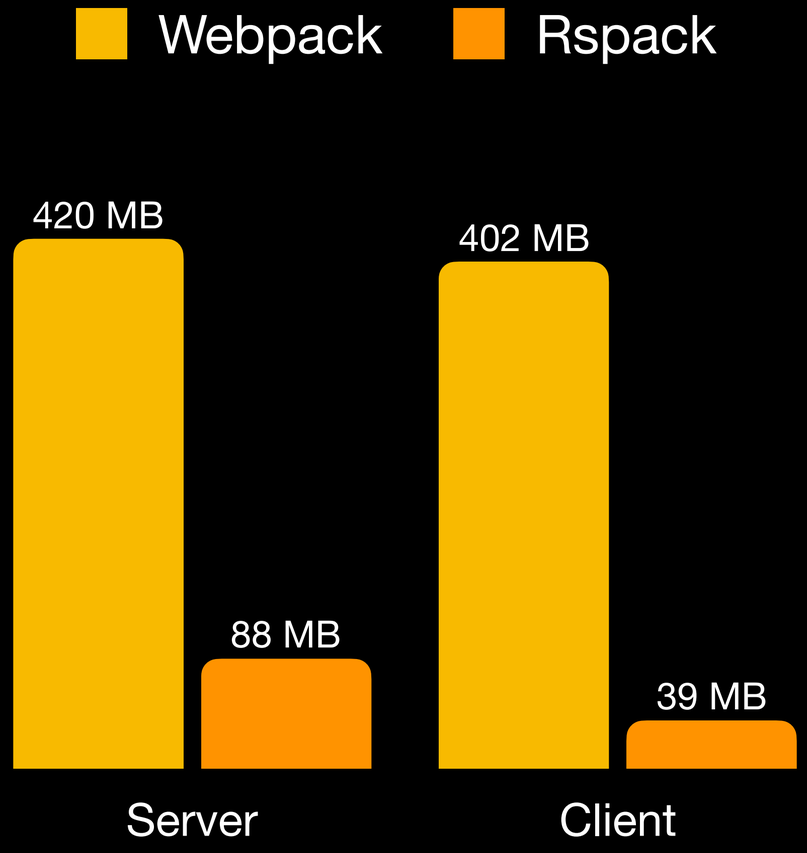
Rspack Demo

Account Dashboard and MyAccount

Fast Build Time



Less Memory Usage



Max Memory Usage in MyAccount
pnpm run build

Q & A

Rust

A language empowering everyone
to build reliable and efficient software.

What’s Rust?

- Systems programming language
- Focus: performance, safety, concurrency
- Developed by Mozilla (2010)

Programming Language Comparison

Feature	Rust	Node.js
Memory Management	Ownership & Borrowing	Garbage Collection
Static Typing	Static	Dynamic
Performance	Highest (Comparable to C++)	Medium (Modern Engines)
Compile-Time Checks	Extensive	Minimal
Concurrency	Built-in Safe	Single-threaded (Async I/O)
Runtime Overhead	Minimal	Significant
Primary Use Cases	Systems, WebAssembly	Web, Back-end Development
Learning Curve	Steep	Easy

Programming Language Comparison

Feature	C++	Rust	Node.js	Python
Memory Management	Manual (RAII, Smart Pointers)	Ownership & Borrowing	Garbage Collection	Garbage Collection
Static Typing	Static	Static	Dynamic	Dynamic (Optional Typing)
Performance	Highest	Highest (Comparable to C++)	Medium (Modern Engines)	Low to Medium
Compile-Time Checks	Strong (Templates, Compiler)	Extensive	Minimal	Minimal (With Type Hinting)
Concurrency	Complex (Low-Level APIs)	Built-in Safe	Single-threaded (Async I/O)	Multi-threading (GIL-Limited)
Runtime Overhead	Minimal	Minimal	Significant	Moderate
Primary Use Cases	Systems/Game Development	Systems, WebAssembly	Web, Back-end Development	Data Science, Web, Automation
Learning Curve	Steep	Steep	Easy	Easy

Ownership in Rust

- Each value has its own single owner
- When owner goes out of scope, value is automatically dropped
- Ownership can be transferred, but not duplicated without explicit cloning

Ownership in Rust

🔒 Ownership = Single Key to a Resource

- Only one key can access a specific resource at a time
- Key can be passed, but original owner loses access
- Compiler ensures no unauthorized access

Ownership in Rust

```rust
fn main() {
    // Ownership Transfer
    let hello = String::from("hello");
    let movedHello = hello; // Ownership moves, hello is no longer valid



}
```

Ownership in Rust

```rust
fn main() {
    // Ownership Transfer
    let hello = String::from("hello");
    let movedHello = hello; // Ownership moves, hello is no longer valid

    // Borrowing
    let world = String::from("world");
    let len = calculate_length(&world); // Borrowing without transferring ownership
}
```

螃蟹幼幼班

Rust 入門指南 by Blue Lan

Concurrency Architecture in Rust

- Key Concurrency Primitives
  - std::thread: Native Thread Creation
  - mpsc (Multiple Producer, Single Consumer) Channels
  - Mutex and RwLock for Synchronized Access
  - Arc (Atomic Reference Counting) for Shared Ownership

Concurrency Architecture in Rust

```rust
use std::{mem, thread};

// Fast set `src` into the referenced `dest`, and drop the old value in other thread
// This method is used when the dropping time is long
pub fn fast_set<T>(dest: &mut T, src: T)
where
  T: Send + 'static,
{
  let old = mem::replace(dest, src);
  thread::spawn(move || {
    mem::drop(old);
  });
}
```

Concurrency Architecture in Rust

```rust
pub fn with_rspack_error_handler<F, Ret>(
    title: String, kind: DiagnosticKind,
    cm: Arc<SourceMap>, op: F,
  ) -> std::result::Result<Ret, BatchErrors>
  where
    F: FnOnce(&Handler) -> std::result::Result<Ret, BatchErrors>,
  {
    let (tx, rx) = mpsc::channel();
    let emitter = RspackErrorEmitter {
      title,
      kind,
      source_map: cm,
      tx,
    };









  }
```

Concurrency Architecture in Rust

```rust
pub fn with_rspack_error_handler<F, Ret>(
    title: String, kind: DiagnosticKind,
    cm: Arc<SourceMap>, op: F,
  ) -> std::result::Result<Ret, BatchErrors>
  where
    F: FnOnce(&Handler) -> std::result::Result<Ret, BatchErrors>,
  {
    let (tx, rx) = mpsc::channel();
    let emitter = RspackErrorEmitter {
      title,
      kind,
      source_map: cm,
      tx,
    };

    let handler = Handler::with_emitter(true, false, Box::new(emitter));
    let ret = HANDLER.set(&handler, || op(&handler));









  }
```

Concurrency Architecture in Rust

```rust
pub fn with_rspack_error_handler<F, Ret>(
    title: String, kind: DiagnosticKind,
    cm: Arc<SourceMap>, op: F,
  ) -> std::result::Result<Ret, BatchErrors>
  where
    F: FnOnce(&Handler) -> std::result::Result<Ret, BatchErrors>,
  {
    let (tx, rx) = mpsc::channel();
    let emitter = RspackErrorEmitter {
      title,
      kind,
      source_map: cm,
      tx,
    };

    let handler = Handler::with_emitter(true, false, Box::new(emitter));
    let ret = HANDLER.set(&handler, || op(&handler));

    if handler.has_errors() {
      drop(handler);
      Err(BatchErrors(rx.into_iter().collect()))
    } else {
      ret
    }
  }
```

Concurrency Architecture in Rust

```rust
struct ThreadsafeJsValueRefHandle<T: NapiValue> {
  value_ref: Arc<Mutex<JsValueRef<T>>>,
  drop_handle: JsCallback<Box<dyn FnOnce(Env)>>,
}

impl<T: NapiValue> ThreadsafeJsValueRefHandle<T> {
  fn new(env: Env, js_ref: JsValueRef<T>) -> Result<Self> {
    Ok(Self {
      value_ref: Arc::new(Mutex::new(js_ref)),
      drop_handle: unsafe { JsCallback::new(env.raw()) }?,
    })
  }
}
```

Concurrency Architecture in Rust

```rust
struct ThreadsafeJsValueRefHandle<T: NapiValue> {
  value_ref: Arc<Mutex<JsValueRef<T>>>,
  drop_handle: JsCallback<Box<dyn FnOnce(Env)>>,
}

impl<T: NapiValue> ThreadsafeJsValueRefHandle<T> {
  fn new(env: Env, js_ref: JsValueRef<T>) -> Result<Self> {
    Ok(Self {
      value_ref: Arc::new(Mutex::new(js_ref)),
      drop_handle: unsafe { JsCallback::new(env.raw()) }?,
    })
  }
}

impl<T: NapiValue> Drop for ThreadsafeJsValueRefHandle<T> {
  fn drop(&mut self) {
    let value_ref = self.value_ref.clone();
    self.drop_handle.call(Box::new(move |env| {
      let _ = value_ref
        .lock()
        .expect("should lock `value_ref`")
        .unref(env);
    }))
  }
}
```

Rayon: Parallel Iterators for Rust

```rust
pub fn runtime_modules_code_generation(&mut self) -> Result<()> {
  self.runtime_modules_code_generation_source = self
    .runtime_modules
    .par_iter()
    .map(|(runtime_module_identifier, runtime_module)| -> Result<_> {
      let result = runtime_module.code_generation(self, None, None)?;
      let source = result
        .get(&SourceType::Runtime)
        .expect("should have source");
      Ok((*runtime_module_identifier, source.clone()))
    })
    .collect::<Result<_>>()?;
  self
    .code_generated_modules
    .extend(self.runtime_modules.keys().copied());
  Ok(())
}
```

Rayon: Parallel Iterators for Rust

- Parallel Iterators
  - Transforms regular iterators into parallel ones
  - Enables parallel operations like map, filter, and reduce
- Work Stealing
  - Efficiently distributes work among threads
  - Ensures that all threads are kept busy

Rayon: Parallel Iterators for Rust

- Fine-grained Parallelism
  - Can parallelize even small tasks for optimal performance
- Easy to Use
  - Integrates seamlessly with Rust's iterator pattern

Why Rust?

- Safety
  - Prevents memory errors like null pointer dereferencing and buffer overflows via its ownership model
- Performance
  - Compiled to native machine code, delivering performance comparable to C/C++
  - Zero-cost abstractions ensure efficient code without runtime overhead

Why Rust?

- Concurrency
  - Ensures thread safety and prevents data races through its ownership model
  - Native patterns like threads and libraries such as Rayon and Tokio provide powerful tools for parallel and async programming

Development Tool Migration Trends

- JavaScript Ecosystem
  - Build/Bundle: Rspack, Rolldown, Turbopack
  - Compiler: SWC, Oxc
  - Linter / Formatter: Biome, Oxc

Development Tool Migration Trends

- Python Ecosystem
  - Linter / Formatter: Ruff
  - Packaging: PyOxidizer

Rust is the best and fastest language for AST tooling

Q & A

Thank you for listening