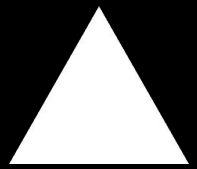


Where was I
a year ago today?

Muted

by Michael Arnaldi

on 𝕏.com

Well, well, well, how the turntables...

Vercel 🤝 Effect

A case study on vercel domains | Effect Days 2025

▲ ~/whois

Dillon Mulroy

I am a software engineer with 11 years of experience, primarily in the JavaScript, TypeScript, NodeJS, and React ecosystem(s). I currently lead Vercel's Domains product area and engineering team.

I have been using Effect since it's 3.0 release and it's become a foundational technology for me in building reliable and scalable applications.

→ x.com/dillon_mulroy

→ twitch.tv/dmmulroy

→ github.com/dmmulroy

▲ ~/agenda

The problem with TypeScript

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)
- Why we’re betting on Effect

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)
- Why we’re betting on Effect
- The power of an Error

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)
- Why we’re betting on Effect
- The power of an Error
- TLD Metadata Service(s)

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)
- Why we’re betting on Effect
- The power of an Error
- TLD Metadata Service(s)
- Takeaways & the future

▲ ~/agenda

- The problem with TypeScript
- Qualities of “production software” (imo)
- Why we’re betting on Effect
- The power of an Error
- TLD Metadata Service(s)
- Takeaways & the future
- Appendix: Useful patterns & code snippets

The problem with TypeScript

Only one?

Only one?

🤔

(just kidding)

▲ ~/the-problem-with-typescript

My hot take is that JS has the lowest bar of entry to building and being productive (a good thing) but one of the highest bars of any language and stack to building high quality, stable, and reliable software.

▲ ~/the-problem-with-typescript

The biggest problem with TypeScript/Node backends is that the language makes it far too easy to only ever consider the happy path

▲ ~/the-problem-with-typescript

The biggest problem with TypeScript/Node backends is that the language makes it far too easy to only ever consider the happy path

▲ ~/the-problem-with-typescript

Happy path blindness

The biggest problem with TypeScript/Node backends is that the language makes it far too easy to only ever consider the happy path

Happy path blindness

```typescript
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

```typescript
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

→ What happens if `Auth.check` throws? **Does it throw?** Can it throw **more than one kind** of error?

```
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

→ What happens if `Auth.check` throws? **Does it throw?** Can it throw **more than one kind** of error?

→ What about `db.query`?

```
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

→ What happens if `Auth.check` throws? **Does it throw?** Can it throw **more than one kind** of error?

→ What about `db.query`?

→ Can we **retry** on any errors?

```
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

→ What happens if `Auth.check` throws? **Does it throw?** Can it throw **more than one kind** of error?

→ What about `db.query`?

→ Can we **retry** on any errors?

→ If so, do we need to consider a **backoff** interval for retrying?

```typescript
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

▲ ~/the-problem-with-typescript

Happy path blindness

→ What happens if `Auth.check` throws? **Does it throw?** Can it throw **more than one kind** of error?

→ What about `db.query`?

→ Can we **retry** on any errors?

→ If so, do we need to consider a **backoff** interval for retrying?

→ How should we **communicate errors to callers**? Should we pass through errors? All of them? Should we wrap them with **custom errors**?

```typescript
function fetchDomainByNameForUser(
  domainName: string,
  userId: string,
): Promise<Domain> {
  await Auth.check({
    action: "query",
    resource: "domains",
    userId,
  });

  const domain = await db.query("domains", { name: domainName });

  if (!domain) {
    throw new Error("Domain not found");
  }

  return domain;
}
```

Happy path blindness

"I'll just deal with errors in the calling code."

- Me, 6 months ago, probably

Happy path blindness

```
export async function GET(
  request: NextRequest,
  { params }: { params: { name: string } },
) {
  const domainName = params.name;
  const userId = getUserIdFromRequest(request);

  const domain = await fetchDomainByNameForUser(domainName, userId);

  return NextResponse.json(domain);
}
```

Happy path blindness

“I know this code works safely for my use case, catching the error at the route level will be fine, it’s just for an internal tool”

- Also me, 6 months ago, probably

Happy path blindness

"I need a function to fetch a domain. This looks useful, I'm sure it's safe to use."

- New developer on the team

Happy path blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

Happy path blindness

```
import { fetchDomai
```

```
om "@/lib/domains"
```

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

Happy path blindness

```
import { fetchDomai
```

```
m "@/lib/domains"
```

```
import { fetchDomainByNameForUse
```

```
om "@/lib/domains"
```

```
import { fetchDomainByNameforUser } from "@/lib/domains"
```

▲ ~/the-problem-with-typescript

Happy path blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

▲ ~/the-problem-with-typescript

Happy path blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

▲ ~/the-problem-with-typescript

Happy path blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

▲ ~/the-problem-with-typescript

Happy path blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

-with-typescript

th blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

-with-typescript

th blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

-with-typescript

th blindness

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

import { fetchDomainByNameForUser } from "@/lib/domains"

```
import { fetchDomainByNameForUser } from "@/lib/domains"
```

▲ ~/the-problem-with-typescript

Only one?

🤔

(just kidding)

Only one?

🤔

(just kidding - only kinda)

▲ ~/the-problem-with-typescript/honorable-mentions

Errors are typed as unknown

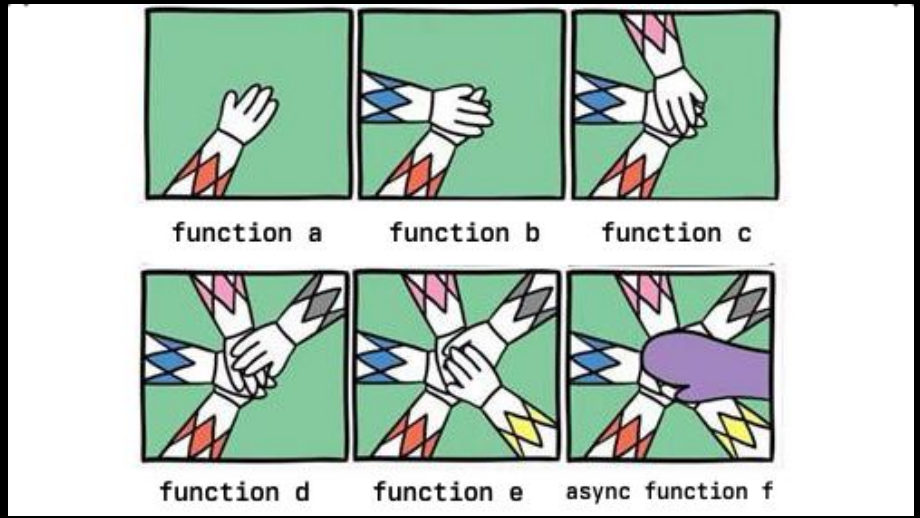
Errors are typed as unknown

```
function getDnsRecords(dnsZone: string): Promise<DnsRecord[]> {}

class DnsPropogationDelayError extends Error {}
class DnsZoneNotFoundError extends Error {}
class InvalidDnsZoneError extends Error {}

function handler(req, res) {
  try {
    // ...
    const dnsRecords = await getDnsRecords("v0.dev");
    //...
  } catch (error) {
    //     ^? `unknown`
  }
}
```

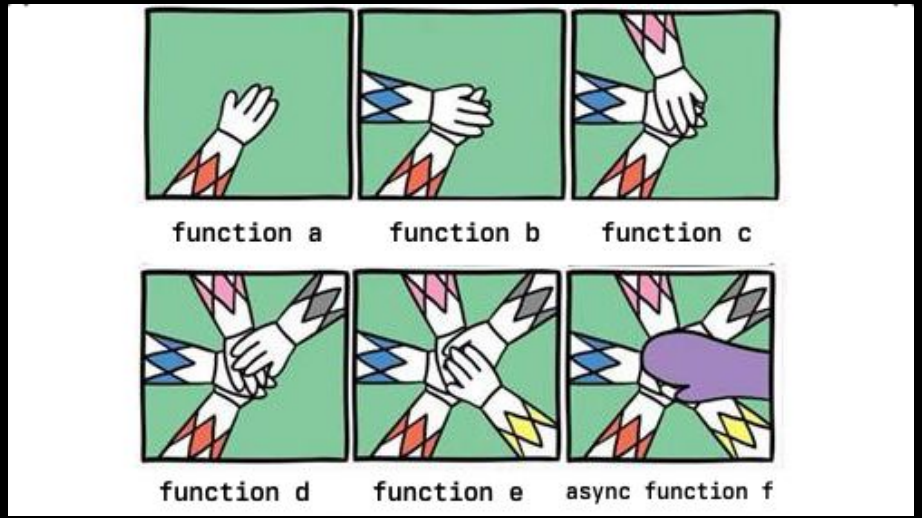
Function coloring

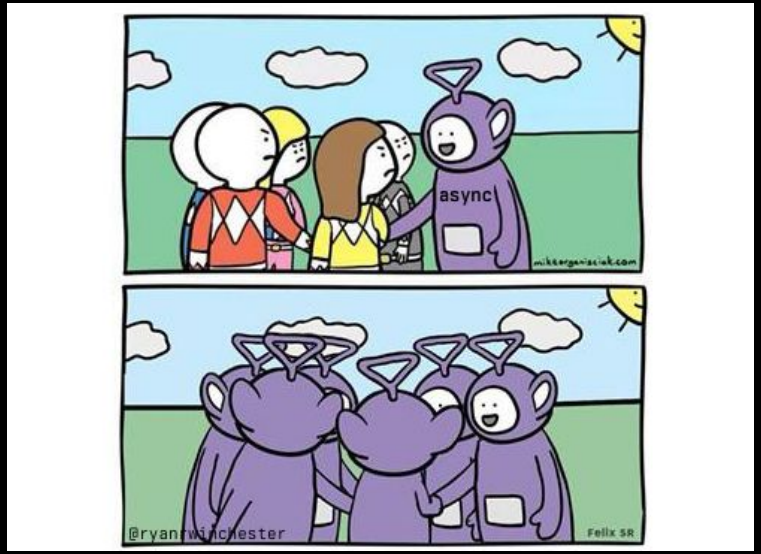
▲ ~/the-problem-with-typescript/honorable-mentions

Function coloring



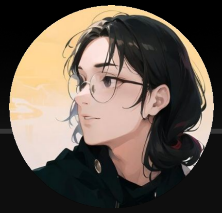
Function coloring



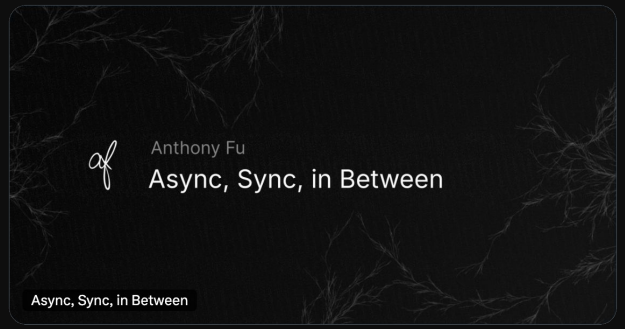


▲ ~/the-problem-with-typescript/honorable-mentions

Function coloring



New blog post! Let's mess around with sync, async, and Schrödinger's Cat! 😼



From antfu.me

Function coloring

```
import { quansync } from "quansync";

export const readFile = quansync({
  sync: (filepath) => fs.readFileSync(filepath),
  async: (filepath) => fs.promises.readFile(filepath),
});

// Create a quansync with `function*` and `yield*`
export const readJSON = quansync(function* (filepath) {
  // Call the quansync function directly
  // and use `yield*` to get the result.
  // Upon usage, it will auto select the implementation
  const content = yield* readFile(filepath);
  return JSON.parse(content);
});
```

Fragmented ecosystem

Fragmented ecosystem

→ We don't have our version of Rails, Laravel, or Spring Boot

Fragmented ecosystem

→ We don't have our version of Rails, Laravel, or Spring Boot

→ Libraries and packages are not built to work together or with similar design goals

Fragmented ecosystem

→ We don't have our version of Rails, Laravel, or Spring Boot

→ Libraries and packages are not built to work together or with similar design goals

→ In large code bases this can result in reinventing the wheel many times over (poorly)

Fragmented ecosystem

→ We don't have our version of Rails, Laravel, or Spring Boot

→ Libraries and packages are not built to work together or with similar design goals

→ In large code bases this can result in reinventing the wheel many times over (poorly)

→ Sindre Sorhus is carrying the ecosystem in his backpack

▲ ~/the-problem-with-typescript/honorable-mentions

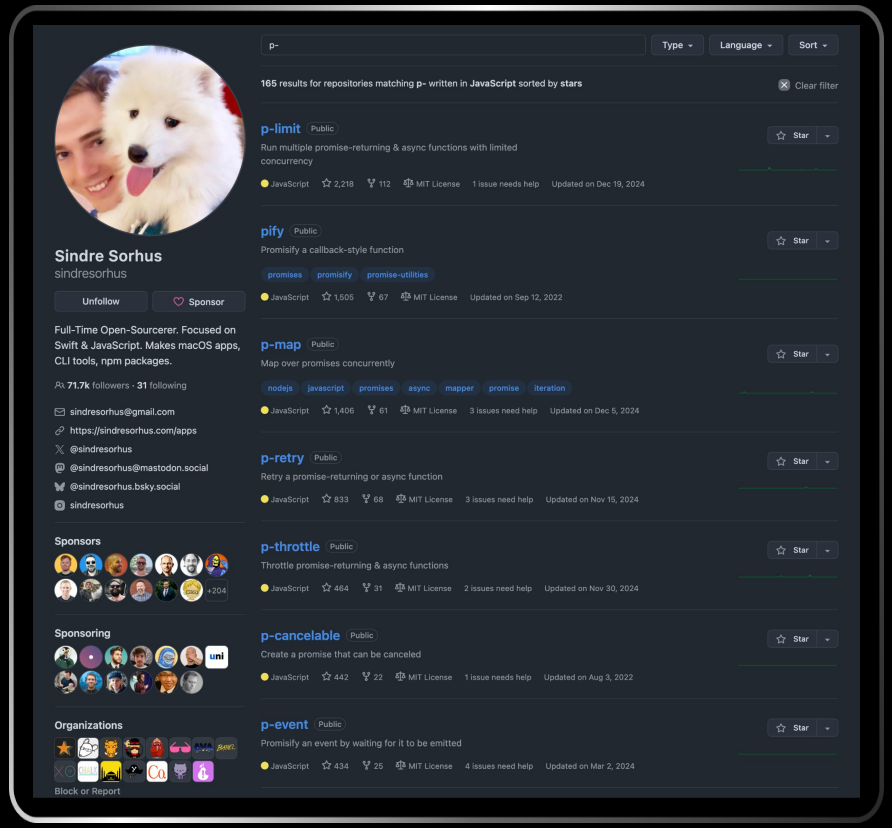
Fragmented ecosystem

→ We don't have our version of Rails, Laravel, or Spring Boot

→ Libraries and packages are not built to work together or with similar design goals

→ In large code bases this can result in reinventing the wheel many times over (poorly)

→ Sindre Sorhus is carrying the ecosystem in his backpack



Qualities of “production software”

Qualities of "production software"

(imo)

▲ ~/qualities-of-production-software

Predictable

▲ ~/qualities-of-production-software

- **Predictable**
- **Observable**

▲ ~/qualities-of-production-software

- **Predictable**
- **Observable**
- **Refactorable**

▲ ~/qualities-of-production-software

- **Predictable**
- **Observable**
- **Refactorable**
- **Testable**

▲ ~/qualities-of-production-software

- **Predictable**
- **Observable**
- **Refactorable**
- **Testable**
- **Intuitive**

▲ ~/qualities-of-production-software

- **Predictable**
- **Observable**
- **Refactorable**
- **Testable**
- **Intuitive**
- **Scalable**

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Refactorable

Testable

Intuitive

Scalable

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Testable

Intuitive

Scalable

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Intuitive

Scalable

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Scalable

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

Architecturally and **organizationally**.

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

Architecturally and **organizationally**.

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

Architecturally and **organizationally**.

→ These traits, properties, and qualities are **deceivingly hard** to achieve in **TypeScript**, **especially as codebases and engineering orgs scale**.

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

Architecturally and **organizationally**.

→ These traits, properties, and qualities are **deceivingly hard** to achieve in **TypeScript**, **especially as codebases and engineering orgs scale**.

→ Teams and organizations require a high degree of **discipline**

▲ ~/qualities-of-production-software

Predictable

Consistent operation, **error handling**, and recovery.

Observable

Easy troubleshooting with logging, metrics, and **telemetry**.

Refactorable

Easy to add features, **read existing code**, and make modifications.

Testable

Meaningful tests are easy to implement and don't feel like a **chore** to add or maintain.

Intuitive

Developers are **naturally guided** towards simple abstractions and doing the "**right**" thing.

Scalable

Architecturally and **organizationally**.

→ These traits, properties, and qualities are **deceivingly hard** to achieve in **TypeScript**, especially as codebases and engineering orgs scale.

→ Teams and organizations require a high degree of **discipline**

→ Discipline on its own **does not scale**

▲ ~/qualities-of-production-software/consulting

I spent the better part of 6.5 years consulting in the JavaScript, TypeScript, Node, and React ecosystem(s). I’ve worked with:

I spent the better part of 6.5 years consulting in the JavaScript, TypeScript, Node, and React ecosystem(s). I've worked with:

- Fortune 500's

I spent the better part of 6.5 years consulting in the JavaScript, TypeScript, Node, and React ecosystem(s). I've worked with:

- Fortune 500's
- Early stage startups

I spent the better part of 6.5 years consulting in the JavaScript, TypeScript, Node, and React ecosystem(s). I've worked with:

- Fortune 500's
- Early stage startups
- FAANGs

Regardless of **organization/team size**, the **skill level** of engineers, or the type of product being built, I have not seen **TypeScript** scale in a **maintainable** way at the **people** & **process level**.

Why we're betting on Effect

▲ ~/betting-on-effect

TL;DR

It makes doing the hard, tedious, and error prone tasks that require discipline, easy, natural, first class, and fun.

Predictable

Predictable

The **Effect** type can **inform** of us of all the **guarantees** and **constraints** of our program(s) and code **at a glance**

Composable & Typed Error Tracking

Composable & Typed Error Tracking

```
function queryDomain(
  domainName: string,
): Effect.Effect<Domain, DomainNotFoundError>;

function enrichDomain(
  domain: Domain,
): Effect.Effect<EnrichedDomain, EnrichmentError>;

function renewDomain(
  enrichedDomain: EnrichedDomain,
): Effect.Effect<RenewedDomain, RenewalFailure>;
```

Composable & Typed Error Tracking

```
function renewDomainHandler(domainName: string) {
  return Effect.gen(function* () {
    const domain = yield* queryDomain(domainName);
    const enrichedDomain = yield* enrichDomain(domain);
    const renewedDomain = yield* renewDomain(enrichedDomain);

    return renewedDomain;
  });
}
```

Composable & Typed Error Tracking

```
function renewDomainAction(
  domainName: string,
): Effect.Effect<
  RenewedDomain,
  DomainNotFoundError | EnrichmentError | RenewalFailure
>;
```

Dependency Injection

Dependency Injection

```
function processUpstreamRegistrarTld(tldName: string) {
  return Effect.gen(function* () {
    const registrar = yield* UpstreamRegistrar;
    const db = yield* DomainsRegistryElectroDb;
    const priceCache = yield* PriceCache;

    const { tld_data } = yield* registrar.tlds.getTldMetadata(tldName);

    yield* Effect.all([db.persist(tld_data), priceCache.persist(tld_data)]);
  });
}
```

Dependency Injection

```
const processTlds = Effect.gen(function* () {
  // ... code excluded for brevity
  yield* processUpstreamRegistrarTld(".ai");
  // ... code excluded for brevity
});

const ProductionDependencies = Layer.empty.pipe(
  Layer.provide(UpstreamRegistrar.Default),
  Layer.provide(DomainsRegistryElectroDb.Default),
  Layer.provide(PriceCache.Default),
);

Effect.runPromise(processTlds.pipe(Effect.provide(ProductionDependencies)));
```

Dependency Injection

```
const processTlds = Effect.gen(function* () {
  // ... code excluded for brevity
  yield* processUpstreamRegistrarTld(".ai");
  // ... code excluded for brevity
});

const TestDependencies = Layer.empty.pipe(
  Layer.provide(UpstreamRegistrar.Test),
  Layer.provide(DomainsRegistryElectroDb.LocalStack),
  Layer.provide(PriceCache.InMemory),
);

Effect.runPromise(processTlds.pipe(Effect.provide(TestDependencies)));
```

You can just `yield*` things

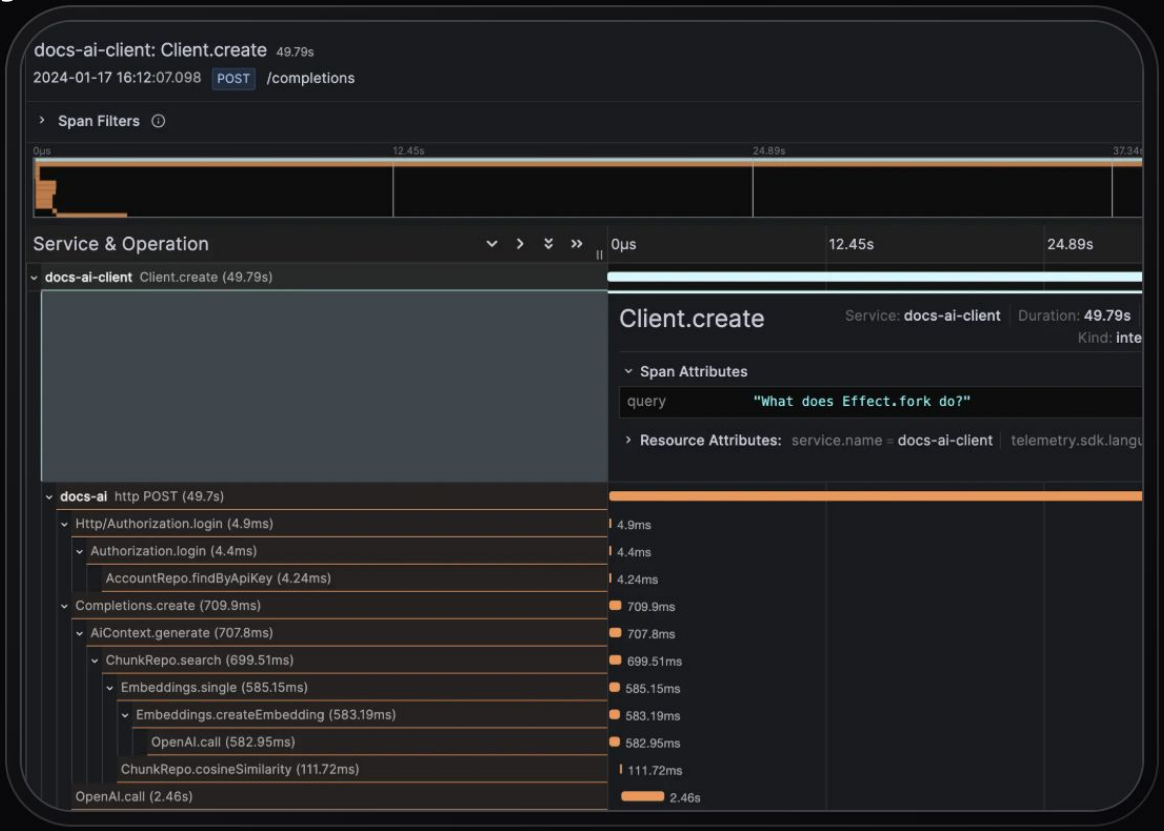
Observability

Observability

```
export const syncTlds = (tlds: Tld[]) => {
  return Effect.gen(function* () {
    // ... implementation ...
  }).pipe(
    Effect.withSpan("upstream_registrar.sync_tlds", {
      attributes: {
        tlds,
      },
    }),
  );
};
```

▲ ~/betting-on-effect

Observability



Harmony

▲ ~/betting-on-effect

Harmony

Case Study

Adopting Effect for Domains

▲ ~/case-study

The Timeline

▲ ~/case-study

The Timeline

December

Self-Serve Renewals

▲ ~/case-study

The Timeline

December

Self-Serve Renewals

January

TLD Metadata Service(s)

▲ ~/case-study

The Timeline

December

Self-Serve Renewals

January

TLD Metadata Service(s)

February

Domains Search/Buy Revamp

▲ ~/case-study

The Timeline

December

Self-Serve Renewals

January

TLD Metadata Service(s)

February

Domains Search/Buy Revamp

March

Effect Days

▲ ~/case-study

The Actual Timeline

▲ ~/case-study

The Actual Timeline

December

Self-Serve Renewals work starts

▲ ~/case-study

The Actual Timeline

December

Self-Serve Renewals work starts

January

Refactor Availability & Buy APIs to support urgent Billing/Money team initiative

▲ ~/case-study

The Actual Timeline

December

Self-Serve Renewals work starts

January

Refactor Availability & Buy APIs to support urgent Billing/Money team initiative

February

Self-Serve Renewals

The Domains Team becomes official. We add two engineers and a designer 🎉

▲ ~/case-study

The Actual Timeline

December

Self-Serve Renewals work starts

January

Refactor Availability & Buy APIs to support urgent Billing/Money team initiative

February

Self-Serve Renewals

The Domains Team becomes official. We add two engineers and a designer 🎉

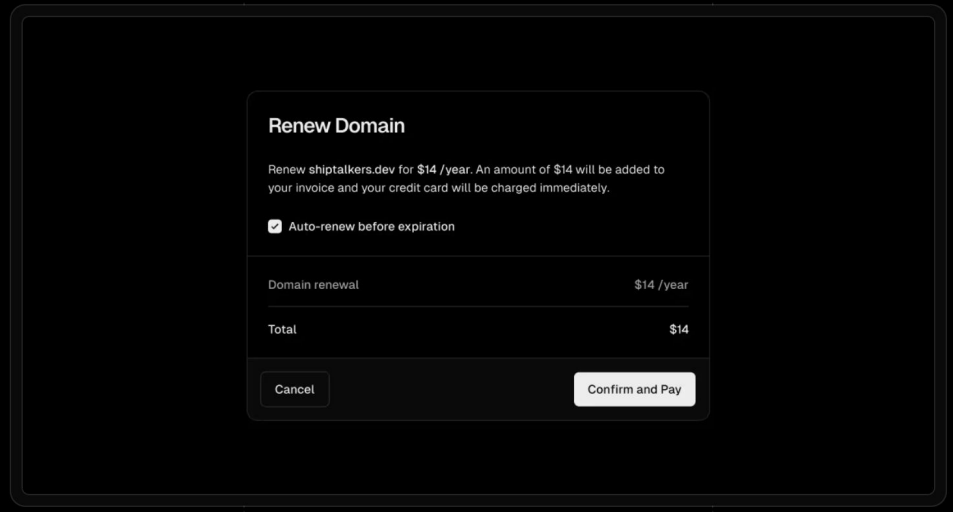
March

TLD Metadata Service ships & Search/Buy planning kick off.

TIMELINES
HD
HISTORY.COM

The power of an Error

Self-serve domain renewals and redemptions now available



▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ Vercel historically only supported **auto-renewals** for domains.

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ Vercel historically only supported **auto-renewals** for domains.

→ The domains product area had the **second highest amount** of CSE tickets month/month, with **a majority resulting from renewal issues**.

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ Vercel historically only supported **auto-renewals** for domains.

→ The domains product area had the **second highest amount** of CSE tickets month/month, with **a majority resulting from renewal issues**.

→ **3 distinct error types**, a deprecated logger, and little to **no telemetry**.

```
type RenewDomainError =
  | ApiError
  | StripePaymentError
  | StripePaymentMethodError
```

Enter

`Data.TaggedError`

```
class DomainsMutexError extends Data.TaggedError("DomainsMutexError")<{
  domainName: string;
  cause?: unknown;
}> {}

const error = new DomainsMutexError({ domainName: "v0.dev" });

error._tag;
//    ^? `DomainsMutexError`
```

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ As part of extracting our renewal from the auto-renewal cron, we **audited our errors** and introduced **neverthrow** and Effect's **Data.TaggedError**

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ As part of extracting our renewal from the auto-renewal cron, we **audited our errors** and introduced **neverthrow** and Effect's **Data.TaggedError**

→ This resulted in a **dramatic reduction** in CSE tickets, **customer empowerment**, and improved time to fix issues/bugs thanks to improved **telemetry**, **tracing**, and **observability**.

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

→ As part of extracting our renewal from the auto-renewal cron, we **audited our errors** and introduced **neverthrow** and Effect's **Data.TaggedError**

→ This resulted in a **dramatic reduction** in CSE tickets, **customer empowerment**, and improved time to fix issues/bugs thanks to improved **telemetry**, **tracing**, and **observability**.

→ We now have **17 distinct error types**. Each with its own context, **user-facing error messages,** internal error messages, constraints, and **retry-ability**

▲ ~/case-study/the-power-of-an-error

Self-Serve Renewals

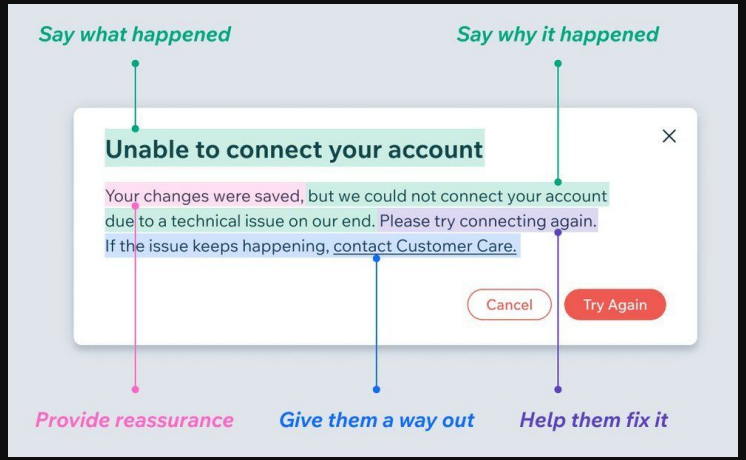
→ As part of extracting our renewal from the auto-renewal cron, we **audited our errors** and introduced **neverthrow** and Effect's **Data.TaggedError**

→ This resulted in a **dramatic reduction** in CSE tickets, **customer empowerment**, and improved time to fix issues/bugs thanks to improved **telemetry**, **tracing**, and **observability**.

→ We now have **17 distinct error types**. Each with its own context, **user-facing error messages,** internal error messages, constraints, and **retry-ability**

```
export type RenewDomainError =
  | CannotRenewError
  | CustomerIdNotFoundError
  | DomainsMutexError
  | DraftInvoiceError
  | ExpirationDateOutOfRangeError
  | GetDomainInvoiceError
  | GetDomainPriceError
  | GetUpstreamRegistrarDomainError
  | InvalidDomainStatusError
  | UpstreamRegistrarDomainNotFoundError
  | UpstreamRegistrarRenewDomainError
  | PayInvoiceError
  | RefundDomainInvoiceError
  | RenewFailureError
  | SyncDomainError
  | TLDConfigNotFoundError
  | UpdateVercelDomainError;
```

WIX
ENGINEERING



TLD Metadata Service(s)

The Problem

▲ ~/case-study/tld-metadata-services

The Problem

→ Our **TLD portfolio**, each TLD's **metadata**, and their **prices** were all managed via a private npm package and various **one off json files**.

▲ ~/case-study/tld-metadata-services

The Problem

→ Our **TLD portfolio**, each TLD's **metadata**, and their **prices** were all managed via a private npm package and various **one off json files**.

→ Making changes required publishing the npm package and **redeploying dozens of services** across **both** our giant TypeScript monorepos.

▲ ~/case-study/tld-metadata-services

The Problem

→ Our **TLD portfolio**, each TLD's **metadata**, and their **prices** were all managed via a private npm package and various **one off json files**.

→ Making changes required publishing the npm package and **redeploying dozens of services** across **both** our giant TypeScript monorepos.

→ Metadata and pricing was **scraped** and or derived from a combination of an **Excel Spreadsheet** (often with human-based data entry errors) and support webpages.

▲ ~/case-study/tld-metadata-services

The Problem

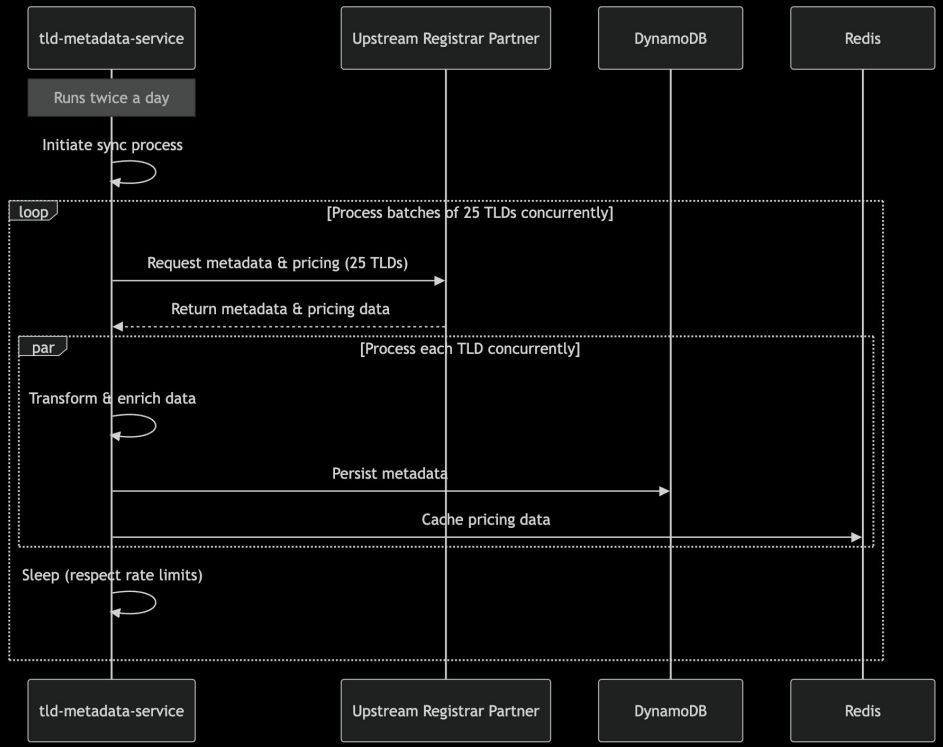
→ Our **TLD portfolio**, each TLD's **metadata**, and their **prices** were all managed via a private npm package and various **one off json files**.

→ Making changes required publishing the npm package and **redeploying dozens of services** across **both** our giant TypeScript monorepos.

→ Metadata and pricing was **scraped** and or derived from a combination of an **Excel Spreadsheet** (often with human-based data entry errors) and support webpages.

▲ ~/case-study/tld-metadata-services

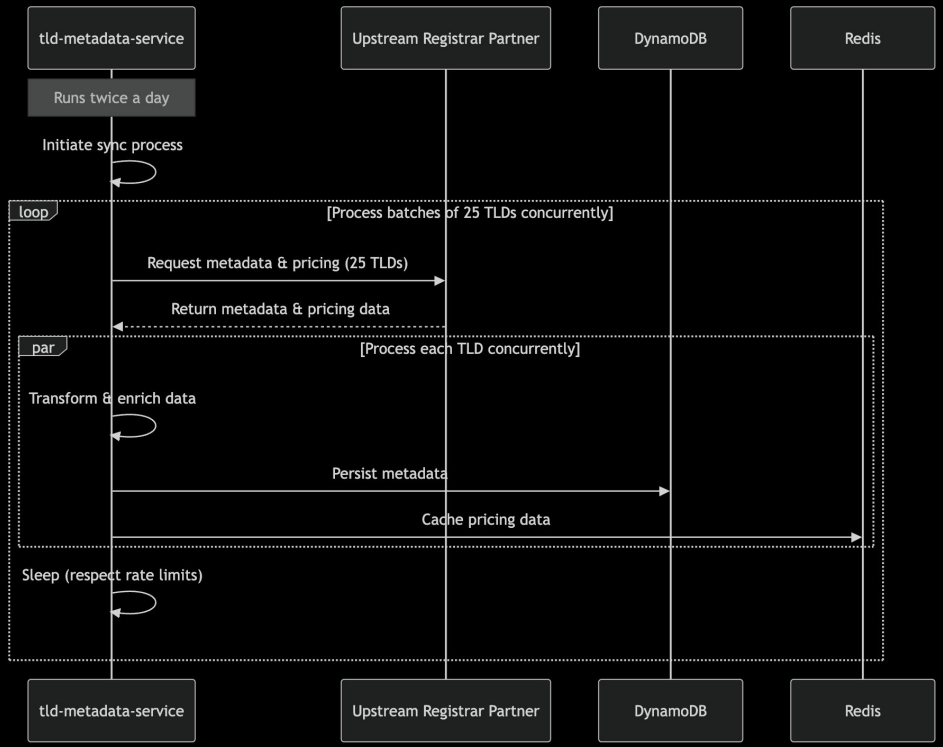
The Solution



▲ ~/case-study/tld-metadata-services

The Solution

→ An Effect services that runs twice daily that **syncs** our entire TLD portfolio's **prices** and **metadata**.

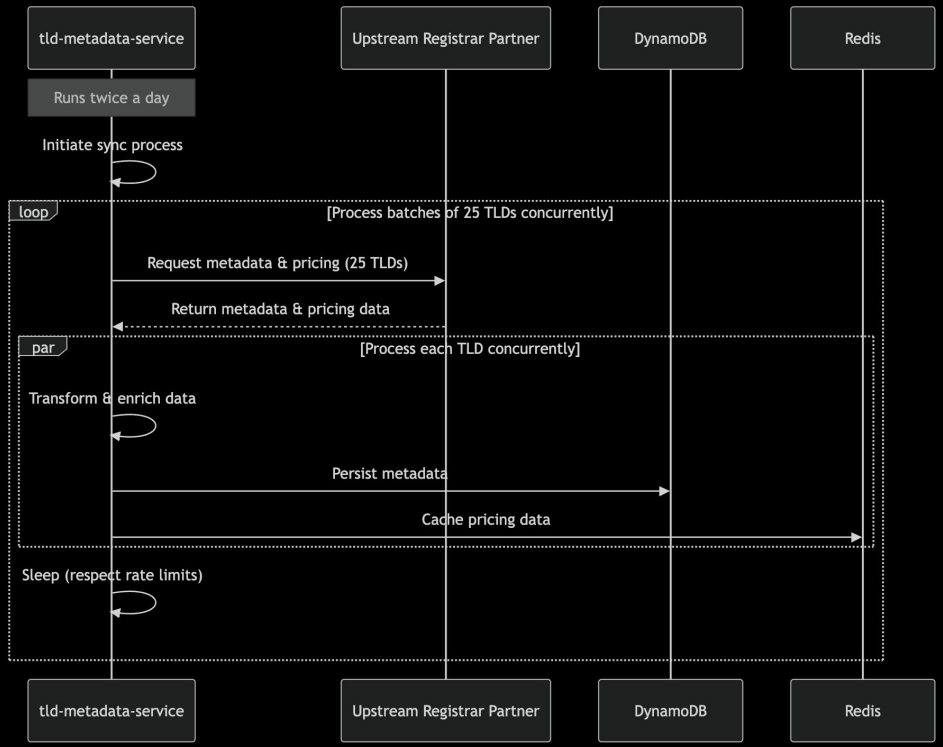


▲ ~/case-study/tld-metadata-services

The Solution

→ An Effect services that runs twice daily that **syncs** our entire TLD portfolio's **prices** and **metadata**.

→ TLD data is processed, enriched, **transformed**, and finally stored in DynamoDB while base TLD prices are cached in Redis.



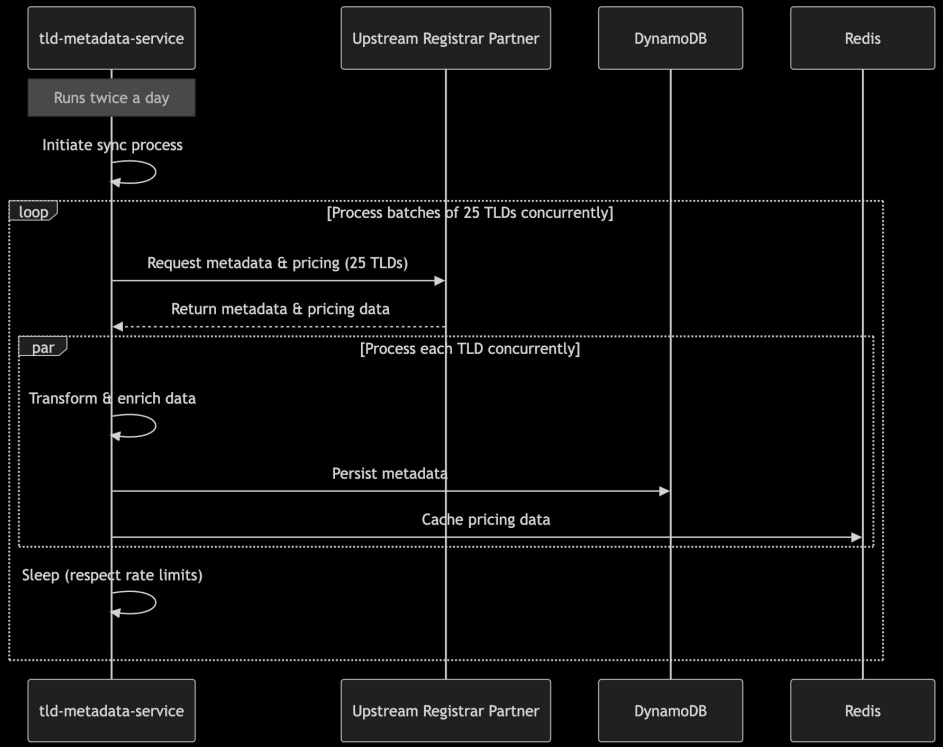
▲ ~/case-study/tld-metadata-services

The Solution

→ An Effect services that runs twice daily that **syncs** our entire TLD portfolio's **prices** and **metadata**.

→ TLD data is processed, enriched, **transformed**, and finally stored in DynamoDB while base TLD prices are cached in Redis.

→ TLD registries are often **unreliable**, inconsistent, and **error prone**. Effect allows us to handle these failures and inconsistencies with **surgical precision**.



▲ ~/case-study/tld-metadata-services

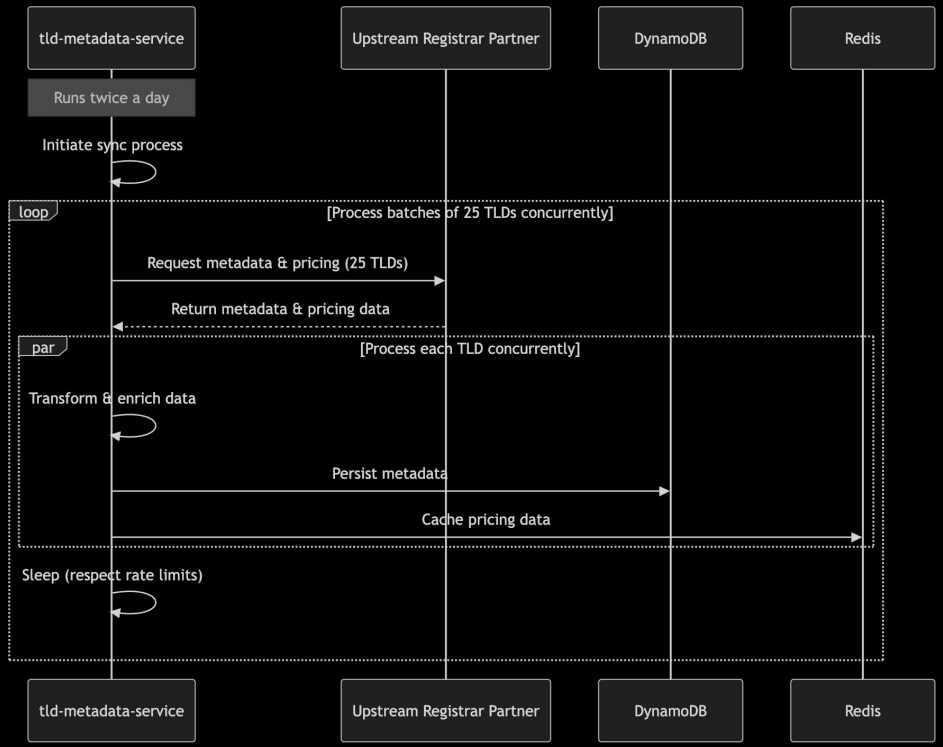
The Solution

→ An Effect services that runs twice daily that **syncs** our entire TLD portfolio's **prices** and **metadata**.

→ TLD data is processed, enriched, **transformed**, and finally stored in DynamoDB while base TLD prices are cached in Redis.

→ TLD registries are often **unreliable**, inconsistent, and **error prone**. Effect allows us to handle these failures and inconsistencies with **surgical precision**.

→ We had to build in such way that this **Network I/O heavy** service would not negatively impact our **rate limits** shared by our **existing services**.



Takeaways & the future

The Good Parts

Vercel 🤝 Effect

▲ ~/take-aways

The Good Parts

→ Answers to the **most common problems** we have at Vercel in our **TypeScript Monorepos**

Vercel

🤝

Effect

▲ ~/take-aways

The Good Parts

→ Answers to the **most common problems** we have at Vercel in our **TypeScript Monorepos**

→ Once the **foundational primitives** are built it's like building with **legos** and you are often able to **ship faster** (with much higher quality!)

Vercel

🤝

Effect

▲ ~/take-aways

The Good Parts

→ Answers to the **most common problems** we have at Vercel in our **TypeScript Monorepos**

→ Once the **foundational primitives** are built it's like building with **legos** and you are often able to **ship faster** (with much higher quality!)

→ Easier to **maintain**, refactor, and I speculate that, possibly unintuitively, it will make **onboarding** and being **productive** an **easier** process.

Vercel

🤝

Effect

▲ ~/take-aways

The Good Parts

→ Answers to the **most common problems** we have at Vercel in our **TypeScript Monorepos**

→ Once the **foundational primitives** are built it's like building with **legos** and you are often able to **ship faster** (with much higher quality!)

→ Easier to **maintain**, refactor, and I speculate that, possibly unintuitively, it will make **onboarding** and being **productive** an **easier** process.

→ The **OpenTelemetry** (OTEL) integration is amazing

Vercel 🤝 Effect

▲ ~/take-aways

The Good Parts

→ Answers to the **most common problems** we have at Vercel in our **TypeScript Monorepos**

→ Once the **foundational primitives** are built it's like building with **legos** and you are often able to **ship faster** (with much higher quality!)

→ Easier to **maintain**, refactor, and I speculate that, possibly unintuitively, it will make **onboarding** and being **productive** an **easier** process.

→ The **OpenTelemetry** (OTEL) integration is **amazing**

→ Schema is **:chef-kiss:**

Vercel 🤝 Effect

The Bad Parts

Vercel

Effect

▲ ~/take-aways

The Bad Parts

→ Buy in can be challenging due to the perceived learning curve

Vercel

🤝

Effect

▲ ~/take-aways

The Bad Parts

→ Buy in can be challenging due to the **perceived learning curve**

→ Getting **OTEL** to play nice with **DataDog** and integrate w/ **existing tracing** was **challenging** (though, this was largely Vercel specific)

Vercel

Effect

▲ ~/take-aways

The Bad Parts

→ Buy in can be challenging due to the **perceived learning curve**

→ Getting **OTEL** to play nice with **DataDog** and integrate w/ **existing tracing** was **challenging** (though, this was largely Vercel specific)

→ There is a **large API surface** (it is TypeScripts missing standard library after all) and when **APIs aren't documented** you have to **rely on the Effect Team** or **Discord**. **AI** also currently **struggles** with Effect.

Vercel

🤝

Effect

▲ ~/take-aways

The Bad Parts

→ Buy in can be challenging due to the **perceived learning curve**

→ Getting **OTEL** to play nice with **DataDog** and integrate w/ **existing tracing** was **challenging** (though, this was largely Vercel specific)

→ There is a **large API surface** (it is TypeScripts missing standard library after all) and when **APIs aren't documented** you have to **rely on the Effect Team** or **Discord**. **AI** also currently **struggles** with Effect.

→ Some APIs and abstractions feel too **"low-level"** and aren't as **intuitive** as the rest of the ecosystem

Vercel

🤝

Effect

▲ ~/the-future

I am very excited, eager, and optimistic about the future of Effect, especially at Vercel.

▲ ~/the-future

I am very excited, eager, and optimistic about the future of Effect, especially at Vercel.

- Exploring Effect Cluster for Domain & Cert Renewal Pipelines

▲ ~/the-future

I am very excited, eager, and optimistic about the future of Effect, especially at Vercel.

- Exploring Effect Cluster for Domain & Cert Renewal Pipelines
- Hoping to start internal lunch & learns and advocating more broadly for adoption

▲ ~/the-future

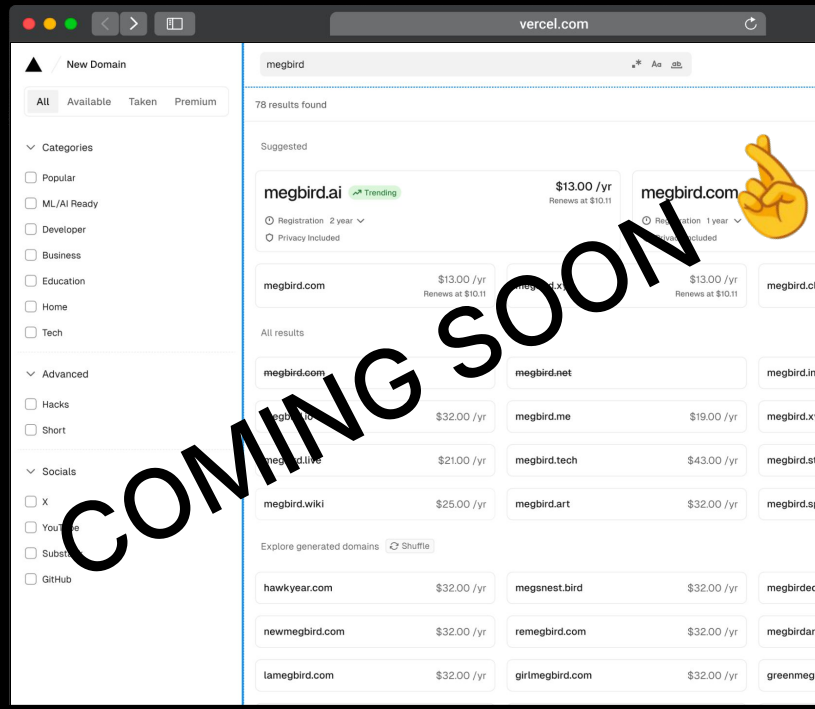
I am very excited, eager, and optimistic about the future of Effect, especially at Vercel.

- Exploring Effect Cluster for Domain & Cert Renewal Pipelines
- Hoping to start internal lunch & learns and advocating more broadly for adoption
- Revamped Search/Buy & Registrar Service(s) powered by Effect

▲ ~/the-future

I am very excited, eager, and optimistic about the future of Effect, especially at Vercel.

- Exploring Effect Cluster for Domain & Cert Renewal Pipelines
- Hoping to start internal lunch & learns and advocating more broadly for adoption
- Revamped Search/Buy & Registrar Service(s) powered by Effect



Thank you

Appendix:
Useful patterns & code snippets

▲ ~/patterns/with-instrumentation

```
getOrdersByDomain(domain, options) {
  return opensrs
    .use((client) =>
      client.lookupCommands.getOrdersByDomain(domain, options),
    )
    .pipe(
      withInstrumentation(
        'registrar.lookup_commands.get_orders_by_domain',
        {
          annotateSpansWith: {
            domain,
          },
          attributes: {
            'order.type': options?.type,
            'order.status': options?.status,
            'order.from': options?.order_from,
            'order.to': options?.order_to,
          },
          retryPolicy: RetryPolicy.exponentialBackoffPolicy,
        },
      ),
    );
},
```

```
export const withInstrumentation =
  (
    spanName: string,
    options: InstrumentationOptions = defaultInstrumentationOptions,
  ) =>
  <Success, Failure, Requirements>(
    effect: Effect.Effect<Success, Failure, Requirements>,
  ) => {
    return Effect.gen(function* () {
      const opts = { ...defaultInstrumentationOptions, ...options };

      const currentSpan = yield* Effect.orElseSucceed(
        Effect.currentSpan,
        () => undefined,
      );

      const logAnnotations = Object.assign(
        {
          'dd.trace_id': currentSpan?.traceId,
          'dd.span_id': currentSpan?.spanId,
        },
        opts.annotateLogsWith,
        opts.annotateSpansWith,
      );

      return yield* effect.pipe(
        Effect.tapError((error) =>
          Effect.annotateCurrentSpan(
            isError(error)
              ? {
                  'error.name': error.name,
                  'error.stack': error.stack,
                  'error.message': error.message,
                  'error.cause': error.cause,
                }
              : flatten(error),
          ),
        ),
        Effect.retry(opts.retryPolicy),
        Effect.annotateLogs(logAnnotations),
        Effect.withSpan(spanName, {
          attributes: opts.attributes,
          kind: opts.spanKind,
          links: opts.spanLinks,
          parent: opts.spanParent,
          root: opts.isRootSpan,
        }),
        Effect.annotateSpans(opts.annotateSpansWith ?? {}),
      );
    });
  };
```

▲ ~/patterns/neverthrow-to-effect

```
/**
 * Converts a `Result` from the `neverthrow` library into an `Effect`.
 *
 * This function transforms a `Result<A, E>` into an `Effect.Effect<A, E, never>` by
pattern matching
 * on the result. If the `Result` is a success, it creates a succeeding `Effect`
with the value.
 * If the `Result` is an error, it creates a failing `Effect` with the error.
 *
 * @template A - The type of the success value.
 * @template E - The type of the error.
 *
 * @param {Result<A, E>} result - The `Result` to convert into an `Effect`.
 * @returns {Effect.Effect<A, E, never>} The resulting `Effect` representing either
success or failure.
 */
export function neverthrowResultToEffect<A, E>(
  result: Result<A, E>,
): Effect.Effect<A, E> {
  return result.match(Effect.succeed, Effect.fail);
}
```

▲ ~/patterns/effect-promise-settled-result

```
/**
 * Converts an Effect into a `PromiseSettledResult`.
 *
 * Runs the provided Effect using the given runtime and converts the result
 * into a settled promise result. If the effect succeeds, it returns a fulfilled
 * result. If it fails, the error is squashed and returned as a rejected result.
 *
 * @param {Runtime.Runtime<never>} runtime - The Effect runtime to execute within.
 * @returns A function that takes an effect and returns a settled promise result.
 */
export const effectToPromiseSettledResult =
  (runtime: Runtime.Runtime<never>) =>
  async <Success,>(
    effect: Effect.Effect<Success>,
  ): Promise<PromiseSettledResult<Success>> => {
    return runtime.runPromiseExit(effect).then((exit) => {
      if (Exit.isSuccess(exit)) {
        return PromiseSettledResult.succeed(exit.value);
      }

      return PromiseSettledResult.fail(Cause.squash(exit.cause));
    });
  };
```