readme.md for @push.rocks/smartmta

A high-performance, enterprise-grade Mail Transfer Agent (MTA) built from scratch in TypeScript with a Rust-powered SMTP engine — no nodemailer, no shortcuts. Automatic MX record discovery means you just call `sendEmail()` and smartmta figures out where to deliver.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Install

```
pnpm install @push.rocks/smartmta
# or
npm install @push.rocks/smartmta
```

After installation, run `pnpm build` to compile the Rust binary (`mailer-bin`). The Rust binary is **required** — `smartmta` will not start without it.

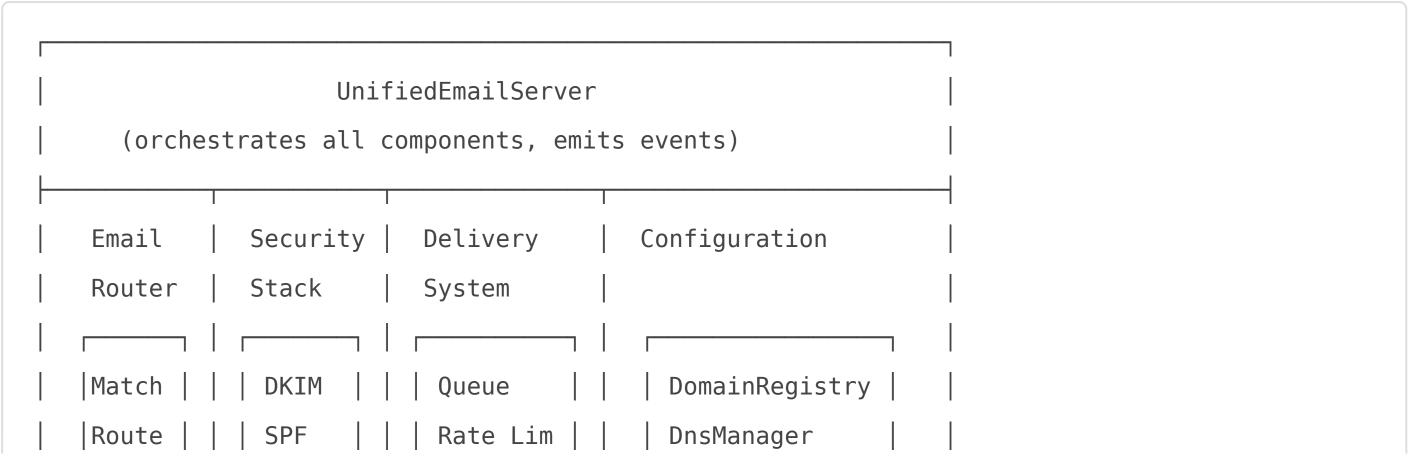
Overview

`@push.rocks/smartmta` is a **complete mail server solution** — SMTP server, SMTP client, email security, content scanning, and delivery management — all built with a custom SMTP implementation. The SMTP engine runs as a Rust binary for maximum performance, communicating with the TypeScript orchestration layer via JSON-over-stdin/stdout IPC.

⚡ What's Inside

Module	What It Does
Rust SMTP Server	High-performance SMTP engine in Rust — TCP/TLS listener, STARTTLS, AUTH, pipelining, per-connection rate limiting
Rust SMTP Client	Outbound delivery with connection pooling, retry logic, TLS negotiation, DKIM signing — all in Rust
DKIM	Key generation, signing, and verification — per domain, with automatic rotation
SPF	Full SPF record validation via Rust
DMARC	Policy enforcement and verification
Email Router	Pattern-based routing with priority, forward/deliver/reject/process actions
Bounce Manager	Automatic bounce detection via Rust, classification (hard/soft), and suppression tracking
Content Scanner	Spam, phishing, malware, XSS, and suspicious link detection — powered by Rust
IP Reputation	DNSBL checks, proxy/TOR/VPN detection, risk scoring via Rust
Rate Limiter	Hierarchical rate limiting (global, per-domain, per-IP)
Delivery Queue	Persistent queue with exponential backoff retry
Template Engine	Email templates with variable substitution
Domain Registry	Multi-domain management with per-domain configuration
DNS Manager	Automatic DNS record management (MX, SPF, DKIM, DMARC)

Architecture



```
│  │ Act  │ │ │ DMARC │ │ │ Retry    │ │  │ DKIMCreator    │   │
│  └──────┘ │ │ IPRep │ │ └─────────┘ │  │ Templates      │   │
│           │ │ Scan  │ │             │  └────────────────┘   │
│           │ └───────┘ │             │                       │
├───────────┴───────────┴─────────────┴───────────────────────┤
│            Rust Security Bridge (smartrust IPC)             │
├─────────────────────────────────────────────────────────────┤
│                  Rust Acceleration Layer                    │
│  ┌──────────────┐  ┌───────────────┐  ┌──────────────────┐  │
│  │ mailer-smtp  │  │mailer-security│  │   mailer-core    │  │
│  │ SMTP Server  │  │DKIM/SPF/DMARC │  │ Types/Validation │  │
│  │ SMTP Client  │  │IP Rep/Content │  │ MIME/Bounce      │  │
│  │ TLS/AUTH     │  │   Scanning    │  │   Detection      │  │
│  └──────────────┘  └───────────────┘  └──────────────────┘  │
└─────────────────────────────────────────────────────────────┘
```

Data flow for inbound mail:

1. Rust SMTP server accepts the connection and handles the full SMTP protocol
2. On `DATA` completion, Rust runs the security pipeline **in-process** (DKIM/SPF/DMARC verification, content scanning, IP reputation check) — zero IPC round-trips
3. Rust emits an `emailReceived` event via IPC with pre-computed security results attached
4. TypeScript processes the email (routing decisions using the pre-computed results, delivery)
5. Rust sends the final SMTP response to the client

Data flow for outbound mail:

1. TypeScript constructs the email and calls `sendEmail()` (defaults to MTA mode)
2. MTA mode automatically resolves MX records for each recipient domain, sorts by priority, and groups recipients for efficient delivery
3. Sends to Rust via IPC — Rust builds the RFC 2822 message, signs with DKIM, and delivers via its SMTP client with connection pooling
4. Result (accepted/rejected recipients, server response) returned to TypeScript

Usage

Setting Up the Email Server

The central entry point is `UnifiedEmailServer`, which orchestrates the Rust SMTP server, routing, security, and delivery:

```
import { UnifiedEmailServer } from '@push.rocks/smartmta';

const emailServer = new UnifiedEmailServer(dcRouterRef, {
  // Ports to listen on (465 = implicit TLS, 25/587 = STARTTLS)
  ports: [25, 587, 465],
  hostname: 'mail.example.com',

  // Multi-domain configuration
  domains: [
    {
      domain: 'example.com',
      dnsMode: 'external-dns',
      dkim: {
        selector: 'default',
        keySize: 2048,
        rotateKeys: true,
        rotationInterval: 90,
      },
      rateLimits: {
        outbound: { messagesPerMinute: 100 },
        inbound: { messagesPerMinute: 200, connectionsPerIp: 20 },
      },
    },
  ],

  // Routing rules (evaluated by priority, highest first)
  routes: [
    {
      name: 'catch-all-forward',
      priority: 10,
      match: {
        recipients: '*@example.com',
      },
      action: {
        type: 'forward',
        forward: {
          host: 'internal-mail.example.com',
```

```
          port: 25,
        },
      },
    },
    {
      name: 'reject-spam-senders',
      priority: 100,
      match: {
        senders: '*@spamdomain.com',
      },
      action: {
        type: 'reject',
        reject: {
          code: 550,
          message: 'Sender rejected by policy',
        },
      },
    },
  ],

  // Authentication settings for the SMTP server
  auth: {
    required: false,
    methods: ['PLAIN', 'LOGIN'],
    users: [{ username: 'outbound', password: 'secret' }],
  },

  // TLS certificates
  tls: {
    certPath: '/etc/ssl/mail.crt',
    keyPath: '/etc/ssl/mail.key',
  },

  maxMessageSize: 25 * 1024 * 1024, // 25 MB
  maxClients: 500,
});

// start() boots the Rust SMTP server, security bridge, DNS records, and delivery queue
await emailServer.start();
```

> **Note:** `start()` will throw if the Rust binary is not compiled. Run `pnpm build` first.

Sending Emails (Automatic MX Discovery)

The recommended way to send email is `sendEmail()`. It defaults to **MTA mode**, which automatically resolves MX records for each recipient domain via DNS — you don't need to know the destination mail server:

```
import { Email, UnifiedEmailServer } from '@push.rocks/smartmta';

// Build an email
const email = new Email({
  from: 'sender@example.com',
  to: ['alice@gmail.com', 'bob@company.org'],
  subject: 'Hello from smartmta! ',
  text: 'Plain text body',
  html: '<h1>Hello!</h1><p>HTML body with <strong>formatting</strong></p>',
  priority: 'high',
  attachments: [
    {
      filename: 'report.pdf',
      content: pdfBuffer,
      contentType: 'application/pdf',
    },
  ],
});

// Send — MTA mode auto-discovers MX servers for gmail.com and company.org
const emailId = await emailServer.sendEmail(email);

// Optionally specify a delivery mode explicitly
const emailId2 = await emailServer.sendEmail(email, 'mta');
```

In MTA mode, smartmta:

- Resolves MX records for each recipient domain (e.g. `gmail.com`, `company.org`)

- Sorts MX hosts by priority (lowest = highest priority per RFC 5321)
- Tries each MX host in order until delivery succeeds
- Falls back to the domain's A record if no MX records exist
- Groups recipients by domain for efficient batch delivery
- Signs outbound mail with DKIM automatically

Delivery Modes

`sendEmail()` accepts a mode parameter that controls how the email is delivered:

```
public async sendEmail(
  email: Email,
  mode: EmailProcessingMode = 'mta',  // 'mta' | 'forward' | 'process'
  route?: IEmailRoute,
  options?: {
    skipSuppressionCheck?: boolean;
    ipAddress?: string;
    isTransactional?: boolean;
  }
): Promise<string>
```

Mode	Description
`mta` (default)	**Auto MX discovery** — resolves MX records via DNS, delivers directly to the recipient's mail server. No relay configuration needed.
`forward`	**Relay delivery** — forwards the email to a configured SMTP host (e.g. an internal mail gateway or third-party relay).
`process`	**Scan + deliver** — runs the content scanning / security pipeline first, then delivers via auto MX resolution.

Direct SMTP Delivery (Low-Level)

For cases where you know the exact target SMTP server (e.g. relaying to a specific host), use the lower-level `sendOutboundEmail()`:

```
// Send directly to a known SMTP server (bypasses MX resolution)
const result = await emailServer.sendOutboundEmail('smtp.example.com', 587, email, {
  auth: { user: 'sender@example.com', pass: 'your-password' },
  dkimDomain: 'example.com',
```

```
  dkimSelector: 'default',
});

console.log(`Accepted: ${result.accepted.join(', ')}`);
console.log(`Response: ${result.response}`);
// -> Accepted: recipient@example.com
// -> Response: 2.0.0 Ok: queued
```

The `sendOutboundEmail` method:

- Automatically resolves DKIM keys from the `DKIMCreator` for the specified domain
- Uses connection pooling in Rust — reuses TCP/TLS connections across sends
- Configurable connection and socket timeouts via `outbound` options on the server

DKIM Signing & Key Management

DKIM key management is handled by `DKIMCreator`, which generates, stores, and rotates keys per domain. Signing is performed automatically by the Rust SMTP client during outbound delivery:

```
import { DKIMCreator } from '@push.rocks/smartmta';

const dkimCreator = new DKIMCreator('/path/to/keys', storageManager);

// Auto-generate keys if they don't exist
await dkimCreator.handleDKIMKeysForDomain('example.com');

// Get the DNS record you need to publish
const dnsRecord = await dkimCreator.getDNSRecordForDomain('example.com');
console.log(dnsRecord);
// -> { type: 'TXT', name: 'default._domainkey.example.com', value: 'v=DKIM1; k=rsa; p=...' }

// Check if keys need rotation
const needsRotation = await dkimCreator.needsRotation('example.com', 'default', 90);
if (needsRotation) {
  const newSelector = await dkimCreator.rotateDkimKeys('example.com', 'default', 2048);
  console.log(`Rotated to selector: ${newSelector}`);
}
```

When `UnifiedEmailServer.start()` is called:

- DKIM keys are generated or loaded for every configured domain

- Signing is applied to all outbound mail via the Rust security bridge
- Key rotation is checked automatically based on your `rotationInterval` config

🛡️ Email Authentication (SPF, DKIM, DMARC)

All verification is powered by the Rust binary. For inbound mail, `UnifiedEmailServer` runs the full security pipeline **automatically** — DKIM, SPF, DMARC, content scanning, and IP reputation in a single Rust pass. Results are attached as headers (`Received-SPF`, `X-DKIM-Result`, `X-DMARC-Result`).

You can also use the individual verifiers directly:

```
import { DKIMVerifier, SpfVerifier, DmarcVerifier } from '@push.rocks/smartmta';

// SPF verification
const spfVerifier = new SpfVerifier();
const spfResult = await spfVerifier.verify(email, senderIP, heloDomain);
// -> { result: 'pass' | 'fail' | 'softfail' | 'neutral' | 'none', domain, ip }

// DKIM verification
const dkimVerifier = new DKIMVerifier();
const dkimResult = await dkimVerifier.verify(rawEmailContent);
// -> [{ is_valid: true, domain: 'example.com', selector: 'default', status: 'pass' }]

// DMARC verification
const dmarcVerifier = new DmarcVerifier();
const dmarcResult = await dmarcVerifier.verify(email, spfResult, dkimResult);
// -> { action: 'pass' | 'quarantine' | 'reject', policy, spfDomainAligned, dkimDomainAligned
}
```

🔀 Email Routing

Pattern-based routing engine with priority ordering and flexible match criteria. Routes are evaluated by priority (highest first):

```
import { EmailRouter } from '@push.rocks/smartmta';

const router = new EmailRouter([
```

```
{
  name: 'admin-mail',
  priority: 100,
  match: {
    recipients: 'admin@example.com',
    authenticated: true,
  },
  action: {
    type: 'deliver',
  },
},
{
  name: 'external-forward',
  priority: 50,
  match: {
    recipients: '*@example.com',
    sizeRange: { max: 10 * 1024 * 1024 }, // under 10MB
  },
  action: {
    type: 'forward',
    forward: {
      host: 'backend-mail.internal',
      port: 25,
      preserveHeaders: true,
    },
  },
},
{
  name: 'process-with-scanning',
  priority: 10,
  match: {
    recipients: '*@*',
  },
  action: {
    type: 'process',
    process: {
      scan: true,
      dkim: true,
      queue: 'normal',
    },
```

```
    },
  },
]);

// Evaluate routes against an email context
const matchedRoute = await router.evaluateRoutes(emailContext);
```

Route Action Types

Action	Description
`forward`	Forward the email to another SMTP server via the Rust SMTP client
`deliver`	Queue for local MTA delivery
`process`	Queue for processing (with optional content scanning and DKIM signing)
`reject`	Reject with a configurable SMTP error code and message

Match Criteria

Criterion	Description
`recipients`	Glob patterns for recipient addresses (`*@example.com`)
`senders`	Glob patterns for sender addresses
`clientIp`	IP addresses or CIDR ranges
`authenticated`	Require authentication status
`headers`	Match specific headers (string or RegExp)
`sizeRange`	Message size constraints (`{ min?, max? }`)
`subject`	Subject line pattern (string or RegExp)
`hasAttachments`	Filter by attachment presence

󰀀 Rate Limiting

Hierarchical rate limiting to protect your server and maintain deliverability:

```
import { Delivery } from '@push.rocks/smartmta';
const { UnifiedRateLimiter } = Delivery;

const rateLimiter = new UnifiedRateLimiter({
```

```
  global: {
    maxMessagesPerMinute: 1000,
    maxRecipientsPerMessage: 500,
    maxConnectionsPerIP: 20,
    maxErrorsPerIP: 10,
    maxAuthFailuresPerIP: 5,
    blockDuration: 600000, // 10 minutes
  },
  domains: {
    'example.com': {
      maxMessagesPerMinute: 100,
      maxRecipientsPerMessage: 50,
    },
  },
});

// Check before sending
const allowed = rateLimiter.checkMessageLimit(
  'sender@example.com',
  '192.168.1.1',
  recipientCount,
  undefined,
  'example.com'
);

if (!allowed.allowed) {
  console.log(`Rate limited: ${allowed.reason}`);
}
```

□□ Bounce Management

Automatic bounce detection (via Rust), classification, and suppression tracking:

```
import { Core } from '@push.rocks/smartmta';
const { BounceManager } = Core;

const bounceManager = new BounceManager();

// Process an SMTP failure
```

```
const bounce = await bounceManager.processSmtpFailure(
  'recipient@example.com',
  '550 5.1.1 User unknown',
  { originalEmailId: 'msg-123' }
);
// -> { bounceType: 'invalid_recipient', bounceCategory: 'hard', ... }

// Check if an address is suppressed due to bounces
const suppressed = bounceManager.isEmailSuppressed('recipient@example.com');

// Manage the suppression list
bounceManager.addToSuppressionList('bad@example.com', 'repeated hard bounces');
bounceManager.removeFromSuppressionList('recovered@example.com');
```

📧 Email Templates

Template engine with variable substitution for transactional and notification emails:

```
import { Core } from '@push.rocks/smartmta';
const { TemplateManager } = Core;

const templates = new TemplateManager({
  from: 'noreply@example.com',
  footerHtml: '<p>&copy; 2026 Example Corp</p>',
});

// Register a template
templates.registerTemplate({
  id: 'welcome',
  name: 'Welcome Email',
  description: 'Sent to new users',
  from: 'welcome@example.com',
  subject: 'Welcome, {{name}}!',
  bodyHtml: '<h1>Welcome, {{name}}!</h1><p>Your account is ready.</p>',
  bodyText: 'Welcome, {{name}}! Your account is ready.',
  category: 'transactional',
});

// Create an Email object from the template
```

```
const email = await templates.createEmail('welcome', {
  to: 'newuser@example.com',
  variables: { name: 'Alice' },
});
```

🌐 DNS Management

When `UnifiedEmailServer.start()` is called, it automatically ensures MX, SPF, DKIM, and DMARC records are in place for all configured domains:

```
const emailServer = new UnifiedEmailServer(dcRouterRef, {
  hostname: 'mail.example.com',
  domains: [
    {
      domain: 'example.com',
      dnsMode: 'external-dns', // managed via Cloudflare API
    },
  ],
  // ... other config
});

// DNS records are set up automatically on start:
//   - MX records pointing to your mail server
//   - SPF TXT records authorizing your server IP
//   - DKIM TXT records with public keys from DKIMCreator
//   - DMARC TXT records with your policy
await emailServer.start();
```

🦀 Rust Acceleration Layer

Performance-critical operations are implemented in Rust and communicate with the TypeScript runtime via `@push.rocks/smartrust` (JSON-over-stdin/stdout IPC). The Rust workspace lives at `rust/` with four crates:

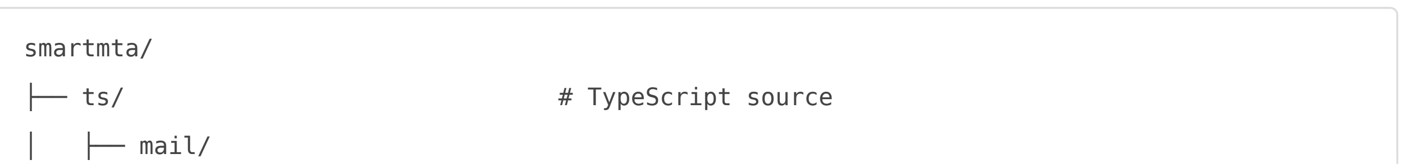
Crate	Status	Purpose
`mailer-core`	✅ Complete (26 tests)	Email types, validation, MIME building, bounce detection

Crate	Status	Purpose
`mailer-security`	Complete (22 tests)	DKIM sign/verify, SPF, DMARC, IP reputation/DNSBL, content scanning
`mailer-smtp`	Complete (106 tests)	Full SMTP protocol engine — TCP/TLS server + client, STARTTLS, AUTH, pipelining, connection pooling, in-process security pipeline
`mailer-bin`	Complete	CLI + smartrust IPC bridge — wires everything together

What Runs Where

Operation	Runs In	Why
SMTP server (port listening, protocol, TLS)	Rust	Performance, memory safety, zero-copy parsing
SMTP client (outbound delivery, connection pooling)	Rust	Connection management, TLS negotiation
DKIM signing & verification	Rust	Crypto-heavy, benefits from native speed
SPF validation	Rust	DNS lookups with async resolver
DMARC policy checking	Rust	Integrates with SPF/DKIM results
IP reputation / DNSBL	Rust	Parallel DNS queries
Content scanning (text patterns)	Rust	Regex engine performance
Bounce detection (pattern matching)	Rust	Regex engine performance
Email validation & MIME building	Rust	Parsing performance
Email routing & orchestration	TypeScript	Business logic, flexibility
Delivery queue & retry	TypeScript	State management, persistence
Template rendering	TypeScript	String interpolation
Domain & DNS management	TypeScript	API integrations

Project Structure



```
smartmta/
├── ts/                                 # TypeScript source
│   ├── mail/
```

```
│   │   ├── core/                    # Email, EmailValidator, BounceManager, TemplateManager
│   │   ├── delivery/                # DeliveryQueue, DeliverySystem, RateLimiter
│   │   ├── routing/                 # UnifiedEmailServer, EmailRouter, DomainRegistry,
DnsManager
│   │   └── security/                # DKIMCreator, DKIMVerifier, SpfVerifier, DmarcVerifier
│   └── security/                    # ContentScanner, IPReputationChecker, RustSecurityBridge
├── rust/                            # Rust workspace
│   └── crates/
│       ├── mailer-core/             # Email types, validation, MIME, bounce detection
│       ├── mailer-security/         # DKIM, SPF, DMARC, IP reputation, content scanning
│       ├── mailer-smtp/             # Full SMTP server + client (TCP/TLS, rate limiting,
pooling)
│       └── mailer-bin/              # CLI + smartrust IPC bridge
├── test/                            # Test suite (116 TypeScript + 154 Rust tests)
├── dist_ts/                         # Compiled TypeScript output
└── dist_rust/                       # Compiled Rust binaries
```

🧪 Testing

The project has comprehensive test coverage with both unit and end-to-end tests:

```
# Build Rust binary first
pnpm build

# Run all tests
pnpm test

# Run specific test files
tstest test/test.e2e.server-lifecycle.node.ts --verbose --timeout 60
tstest test/test.e2e.inbound-smtp.node.ts --verbose --timeout 60
tstest test/test.e2e.routing-actions.node.ts --verbose --timeout 60
tstest test/test.e2e.outbound-delivery.node.ts --verbose --timeout 60
```

E2E tests exercise the full pipeline — starting `UnifiedEmailServer`, connecting via raw TCP sockets, sending SMTP transactions, verifying routing actions, and testing outbound delivery through a mock SMTP receiver.

API Reference

Exported Classes (top-level)

Class	Description
`UnifiedEmailServer`	Main entry point — orchestrates SMTP server, routing, security, and delivery
`Email`	Email message class with validation, attachments, headers, and RFC 822 serialization
`EmailRouter`	Pattern-based route matching and evaluation engine
`DomainRegistry`	Multi-domain configuration manager
`DnsManager`	Automatic DNS record management
`DKIMCreator`	DKIM key generation, storage, rotation
`DKIMVerifier`	DKIM signature verification (delegates to Rust)
`SpfVerifier`	SPF record validation (delegates to Rust)
`DmarcVerifier`	DMARC policy enforcement (delegates to Rust)

Namespaced Exports

Namespace	Classes
`Core`	`Email`, `EmailValidator`, `TemplateManager`, `BounceManager`
`Delivery`	`UnifiedDeliveryQueue`, `MultiModeDeliverySystem`, `DeliveryStatus`, `UnifiedRateLimiter`

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #3
Created 2026-03-28 11:13:40 UTC by foss.global Team
Updated 2026-03-28 12:20:26 UTC by foss.global Team