main 6 Branches 0 Tags Go to file t Go to file Add file Code

YounesElhjouji Merge pull request #8 from Overshoot-ai/dev 148faf0 · 9 hours ago

src	udpated username and credential	10 hours ago
.gitignore	packaged sdk	last month
.npmignore	more like a full package	last month
README.md	updated package name in readme	9 hours ago
package-lock.json	handle media upload in safari	2 weeks ago
package.json	added max tokens	18 hours ago
tsconfig.json	packaged sdk	last month
tsup.config.ts	packaged sdk	last month
vitest.config.ts	packaged sdk	last month

README

Overshoot SDK

> **Warning: Alpha Release**: This is an alpha version (2.0.0-alpha.2). The API may change in future versions.

TypeScript SDK for real-time AI vision analysis on live video streams.

Installation

```
npm install overshoot@alpha
```

Or install a specific alpha version:

```
npm install overshoot@2.0.0-alpha.2
```

Quick Start

> **Note:** The `apiUrl` parameter is optional and defaults to `https://api.overshoot.ai/`. You can omit it for standard usage or provide a custom URL for private deployments.

Camera Source

```
import { RealtimeVision } from "overshoot";

const vision = new RealtimeVision({
  apiKey: "your-api-key-here",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Read any visible text",
  onResult: (result) => {
    console.log(result.result);
    console.log(`Latency: ${result.total_latency_ms}ms`);
  },
});

await vision.start();
```

Video File Source

```
const vision = new RealtimeVision({
  apiKey: "your-api-key-here",
  source: { type: "video", file: videoFile }, // File object from <input type="file">
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect all objects in the video and count them",
  onResult: (result) => {
    console.log(result.result);
  },
});

await vision.start();
```

> **Note:** Video files automatically loop continuously until you call `stop()`.

Screen Capture Source

```
const vision = new RealtimeVision({
  apiKey: "your-api-key-here",
  source: { type: "screen" },
  model: "Qwen/Qwen3-VL-8B-Instruct",
  prompt: "Read any visible text on the screen",
  onResult: (result) => {
    console.log(result.result);
  },
});

await vision.start();
```

> **Note:** Screen capture requires desktop browsers with getDisplayMedia API support. The user will be prompted to select which screen/window to share.

LiveKit Source

If you're on a restrictive network where direct WebRTC connections fail, you can use LiveKit as an alternative video transport. With this source type, you publish video to a LiveKit room yourself, and the SDK handles the server-side stream creation and inference results.

```
const vision = new RealtimeVision({
  apiKey: "your-api-key-here",
  source: {
    type: "livekit",
    url: "wss://your-livekit-server.example.com",
    token: "your-livekit-token",
  },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Describe what you see",
  onResult: (result) => {
    console.log(result.result);
  },
});
```

```
await vision.start();
```

> **Note:** With a LiveKit source, the SDK does not create a local media stream or WebRTC peer connection. You are responsible for publishing video to the LiveKit room using the LiveKit client SDK. The `getMediaStream()` method will return `null` for LiveKit sources.

Configuration

RealtimeVisionConfig

```
interface RealtimeVisionConfig {
  // Required
  apiKey: string; // API key for authentication
  source: StreamSource; // Video source (see StreamSource below)
  model: string; // Model name (see Available Models below)
  prompt: string; // Task description for the model
  onResult: (result: StreamInferenceResult) => void;

  // Optional
  apiUrl?: string; // API endpoint (default: "https://api.overshoot.ai/")
  backend?: "overshoot" | "gemini"; // Model backend (default: "overshoot")
  mode?: "clip" | "frame"; // Processing mode (see Processing Modes below)
  outputSchema?: Record<string, any>; // JSON schema for structured output
  maxOutputTokens?: number; // Cap tokens per inference request (see below)
  onError?: (error: Error) => void;
  debug?: boolean; // Enable debug logging (default: false)

  // Clip mode processing (default mode)
  clipProcessing?: {
    fps?: number; // Source frames per second (1–120, auto-detected for cameras)
    sampling_ratio?: number; // Fraction of frames to process (0–1, default: 0.8)
    clip_length_seconds?: number; // Duration of each clip (0.1–60s, default: 0.5)
    delay_seconds?: number; // Interval between inferences (0–60s, default: 0.2)
  };

  // Frame mode processing
  frameProcessing?: {
    interval_seconds?: number; // Interval between frame captures (0.1–60s, default: 0.2)
  };

  iceServers?: RTCIceServer[]; // Custom WebRTC ICE servers (uses Overshoot TURN servers by default)
}
```

StreamSource

```
type StreamSource =
  | { type: "camera"; cameraFacing: "user" | "environment" }
  | { type: "video"; file: File }
  | { type: "screen" }
  | { type: "livekit"; url: string; token: string };
```

`maxOutputTokens`

Caps the maximum number of tokens the model can generate per inference request. To ensure low latency, the server enforces a limit of **128 effective output tokens per second** per stream:

```
effective_tokens_per_second = max_output_tokens × requests_per_second
```

Where `requests_per_second` is `1 / delay_seconds` (clip mode) or `1 / interval_seconds` (frame mode).

If omitted, the server auto-calculates the optimal value: `floor(128 × interval)`. For example, with `delay_seconds: 0.5` (2 requests/sec), the server defaults to `floor(128 × 0.5) = 64` tokens per request.

If provided, the server validates that `max_output_tokens / interval ≤ 128`. If exceeded, the request is rejected with a 422 error.

Scenario	Interval	Requests/sec	Max allowed `maxOutputTokens`
Clip mode, fast updates	0.2s	5	25
Clip mode, default	0.5s	2	64
Clip mode, slow	1.0s	1	128
Frame mode, default	0.2s	5	25
Frame mode, slow	2.0s	0.5	256
Frame mode, very slow	5.0s	0.2	640

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Describe what you see briefly",
  maxOutputTokens: 100, // Must satisfy: 100 / delay_seconds ≤ 128
  clipProcessing: {
    delay_seconds: 1.0, // 1 request/sec → 100 tokens/sec ≤ 128 ✓
  },
  onResult: (result) => console.log(result.result),
});
```

Available Models

Model	Description
`Qwen/Qwen3-VL-30B-A3B-Instruct`	Very fast and performant general-purpose vision-language model.
`Qwen/Qwen3-VL-8B-Instruct`	Similar latency to 30B. Particularly good at OCR and text extraction tasks.
`OpenGVLab/InternVL3_5-30B-A3B`	Excels at capturing visual detail. More verbose output, higher latency.

Fetching Available Models

Use `StreamClient.getModels()` to query available models and their current status before starting a stream. You can also create a `StreamClient` alongside `RealtimeVision` just for this purpose.

```
import { StreamClient, ModelInfo } from "overshoot";

const client = new StreamClient({ apiKey: "your-api-key" });
const models: ModelInfo[] = await client.getModels();

for (const model of models) {
  console.log(`${model.model}: ${model.status} (ready: ${model.ready})`);
}
```

Each model has a `status` indicating its current load:

Status	`ready`	Meaning	Action
`"ready"`	`true`	Healthy, performing well	Use this model
`"degraded"`	`true`	Near capacity, expect higher latency	Usable, but consider alternatives
`"saturated"`	`false`	At capacity, will reject new streams	Pick a different model
`"unavailable"`	`false`	Endpoint not reachable	Pick a different model

Processing Modes

The SDK supports two processing modes:

Clip Mode (Default)

Processes short video clips with multiple frames, ideal for motion analysis and temporal understanding.

```
const vision = new RealtimeVision({
  // ... other config
  mode: "clip", // Optional - this is the default
  clipProcessing: {
    sampling_ratio: 0.8,        // Process 80% of frames (default)
    clip_length_seconds: 0.5,   // 0.5 second clips (default)
    delay_seconds: 0.2,         // New clip every 0.2s (default)
  },
});
```

Use cases: Activity detection, gesture recognition, motion tracking, sports analysis

Frame Mode

Processes individual frames at regular intervals, ideal for static analysis and fast updates.

```
const vision = new RealtimeVision({
  // ... other config
  mode: "frame",
  frameProcessing: {
    interval_seconds: 0.2, // Capture frame every 0.2s (default)
  },
});
```

Use cases: OCR, object detection, scene description, static monitoring

> **Note:** If you don't specify a mode, the SDK defaults to clip mode. Mode is automatically inferred if you only provide `frameProcessing` config.

Processing Parameters Explained

Clip Mode Parameters

- **`fps`** : The frame rate of your video source. Auto-detected for camera streams; defaults to 30 for video files.
- **`sampling_ratio`** : What fraction of frames to include in each clip (0.8 = 80% of frames, default).
- **`clip_length_seconds`** : Duration of video captured for each inference (default: 0.5 seconds).
- **`delay_seconds`** : How often inference runs (default: 0.2 seconds - 5 inferences per second).

Example with defaults: `fps=30` , `clip_length_seconds=0.5` , `sampling_ratio=0.8` , `delay_seconds=0.2` :

- Each clip captures 0.5 seconds of video (15 frames at 30fps)
- 80% of frames are sampled = 12 frames sent to the model
- New clip starts every 0.2 seconds = ~5 inference results per second

Frame Mode Parameters

- **`interval_seconds`** : Time between frame captures (default: 0.2 seconds - 5 frames per second).

Example with defaults: `interval_seconds=0.2` :

- One frame captured every 0.2 seconds
- ~5 inference results per second

Configuration by Use Case

Different applications need different processing configurations:

Real-time tracking (low latency, frequent updates) - Clip Mode:

```
clipProcessing: {
  sampling_ratio: 0.8,
  clip_length_seconds: 0.5,
```

```
  delay_seconds: 0.2,
}
```

Event detection (monitoring for specific occurrences) - Clip Mode:

```
clipProcessing: {
  sampling_ratio: 0.5,
  clip_length_seconds: 3.0,
  delay_seconds: 2.0,
}
```

Fast OCR/Detection (static analysis) - Frame Mode:

```
mode: "frame",
frameProcessing: {
  interval_seconds: 0.5,
}
```

Structured Output (JSON Schema)

Use `outputSchema` to constrain the model's output to a specific JSON structure. The schema follows JSON Schema specification.

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect objects and return structured data",
  outputSchema: {
    type: "object",
    properties: {
      objects: {
        type: "array",
        items: { type: "string" },
      },
      count: { type: "integer" },
    },
    required: ["objects", "count"],
  },
  onResult: (result) => {
    const data = JSON.parse(result.result);
    console.log(`Found ${data.count} objects:`, data.objects);
  },
});
```

The model will return valid JSON matching your schema. If the model cannot produce valid output, `result.ok` will be `false` and `result.error` will contain details.

> **Note:** `result.result` is always a string. When using `outputSchema`, you must parse it with `JSON.parse()`.

API Methods

```
// Lifecycle
await vision.start(); // Start the video stream
await vision.stop(); // Stop and cleanup resources

// Runtime control
await vision.updatePrompt(newPrompt); // Update task while running

// State access
vision.getMediaStream(); // Get MediaStream for video preview (null if not started)
vision.getStreamId(); // Get current stream ID (null if not started)
vision.isActive(); // Check if stream is running
```

Stream Lifecycle

Keepalive

Streams have a server-side lease (30 second TTL). The SDK automatically sends keepalive requests to renew it. You don't need to manage keepalives manually.

If a keepalive fails (e.g., network issues), the stream will stop and `onError` will be called. If your account runs out of credits, the keepalive returns a 402 error and the stream expires — this is terminal and requires starting a new stream after adding credits.

Network disconnects are permanent. If the client loses connectivity for more than 30 seconds, the lease expires and the stream is destroyed. There is no automatic reconnection — you must call `start()` to create a new stream.

State and Memory

The SDK does not maintain memory or state between inference calls - each frame clip is processed independently. If your application needs to track state over time (e.g., counting repetitions, detecting transitions), implement this in your `onResult` callback:

```
let lastPosition = "up";
let repCount = 0;

const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "user" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect body position: up or down",
  outputSchema: {
    type: "object",
    properties: {
      position: { type: "string", enum: ["up", "down"] }
    },
    required: ["position"]
  },
  onResult: (result) => {
    const data = JSON.parse(result.result);

    // Track state transitions externally
    if (lastPosition === "down" && data.position === "up") {
      repCount++;
      console.log("Rep count:", repCount);
    }
    lastPosition = data.position;
  },
});
```

For result deduplication (e.g., avoiding repeated announcements), track previous results and implement cooldown logic in your application code.

Prompt Engineering

Prompt quality significantly affects results. Here are some tips:

Be specific about output format:

```
prompt: "Count the people visible. Return only a number.";
```

Include examples for complex tasks:

```
prompt: `Describe the primary action happening. Examples:
- "Person walking left"
- "Car turning right"
- "Dog sitting still"`;
```

Request minimal output for lower latency:

```
prompt: "Is there a person in frame? Answer only 'yes' or 'no'.";
```

Provide context when needed:

```
prompt: `You are monitoring a ${locationName}. Alert if you see: ${alertConditions.join(", ")}.`;
```

Use JSON schema for structured data:

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Analyze the scene",
  outputSchema: {
    type: "object",
    properties: {
      description: { type: "string" },
      alert: { type: "boolean" }
    },
    required: ["description", "alert"]
  },
  onResult: (result) => {
    const data = JSON.parse(result.result);
    if (data.alert) {
      console.log("⚠️ Alert:", data.description);
    }
  }
});
```

> **Note:** Prompt effectiveness varies by model. Test different approaches to find what works best for your use case.

React Integration

When using the SDK in React applications, ensure proper cleanup:

```
import { useEffect, useRef, useState } from "react";
import { RealtimeVision } from "overshoot";

function VisionComponent() {
  const visionRef = useRef<RealtimeVision | null>(null);
  const videoRef = useRef<HTMLVideoElement>(null);
  const [isRunning, setIsRunning] = useState(false);

  const startVision = async () => {
    const vision = new RealtimeVision({
      apiKey: "your-api-key",
      source: { type: "camera", cameraFacing: "user" },
      model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
      prompt: "Describe what you see",
      onResult: (result) => {
        console.log(result.result);
      },
      onError: (error) => {
        console.error("Vision error:", error);
        setIsRunning(false);
      },
    });

    await vision.start();
    visionRef.current = vision;
    setIsRunning(true);

    // Attach stream to video element for preview
    const stream = vision.getMediaStream();
    if (stream && videoRef.current) {
      videoRef.current.srcObject = stream;
      videoRef.current.play().catch(console.error);
```

```
    }
  };

  // Cleanup on unmount
  useEffect(() => {
    return () => {
      visionRef.current?.stop();
    };
  }, []);

  return (
    <div>
      <video ref={videoRef} autoPlay playsInline muted />
      <button onClick={startVision} disabled={isRunning}>
        Start
      </button>
      <button onClick={() => visionRef.current?.stop()} disabled={!isRunning}>
        Stop
      </button>
    </div>
  );
}
```

Advanced: Custom Video Sources with StreamClient

For advanced use cases like streaming from a canvas, screen capture, or other custom sources, use `StreamClient` directly:

```
import { StreamClient } from "overshoot";

const client = new StreamClient({
  apiKey: "your-api-key",
});

// Get stream from any source (canvas, screen capture, etc.)
const canvas = document.querySelector("canvas");
const stream = canvas.captureStream(30);
const videoTrack = stream.getVideoTracks()[0];

// Set up WebRTC connection
const peerConnection = new RTCPeerConnection({ iceServers: [...] }); // See default ice servers in RealtimeVison.ts
peerConnection.addTrack(videoTrack, stream);

const offer = await peerConnection.createOffer();
await peerConnection.setLocalDescription(offer);

// Create stream on server
const response = await client.createStream({
  source: { type: "webrtc", sdp: peerConnection.localDescription.sdp },
  mode: "clip",
  processing: {
    sampling_ratio: 0.8,
    fps: 30,
    clip_length_seconds: 0.5,
    delay_seconds: 0.2
  },
  inference: {
    prompt: "Analyze the content",
    backend: "overshoot",
    model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
    max_output_tokens: 25, // Optional: must satisfy max_output_tokens / delay_seconds ≤ 128
  },
});

if (response.webrtc) {
  await peerConnection.setRemoteDescription(response.webrtc);
}

// Connect WebSocket for results
const ws = client.connectWebSocket(response.stream_id);
ws.onopen = () => ws.send(JSON.stringify({ api_key: "your-api-key" }));
```

```
ws.onmessage = (event) => {
  const result = JSON.parse(event.data);
  console.log("Result:", result);
};
```

Examples

Object Detection with Structured Output

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect objects and return JSON: {objects: string[], count: number}",
  outputSchema: {
    type: "object",
    properties: {
      objects: { type: "array", items: { type: "string" } },
      count: { type: "integer" },
    },
    required: ["objects", "count"],
  },
  onResult: (result) => {
    const data = JSON.parse(result.result);
    console.log(`Found ${data.count} objects:`, data.objects);
  },
});

await vision.start();
```

Text Recognition (OCR)

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-8B-Instruct", // 8B is excellent for OCR
  prompt: "Read all visible text in the image",
  mode: "frame", // Frame mode is great for OCR
  onResult: (result) => {
    console.log("Text:", result.result);
  },
});

await vision.start();
```

Dynamic Prompt Updates

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Count people",
  onResult: (result) => console.log(result.result),
});

await vision.start();

// Change task without restarting stream
await vision.updatePrompt("Detect vehicles instead");
```

Debug Mode

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
```

```
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect objects",
  debug: true, // Enable detailed logging
  onResult: (result) => console.log(result.result),
});

await vision.start();
// Console will show detailed connection and processing logs
```

Error Handling

```
const vision = new RealtimeVision({
  apiKey: "your-api-key",
  source: { type: "camera", cameraFacing: "environment" },
  model: "Qwen/Qwen3-VL-30B-A3B-Instruct",
  prompt: "Detect objects",
  onResult: (result) => {
    if (result.ok) {
      console.log("Success:", result.result);
    } else {
      console.error("Inference error:", result.error);
    }
  },
  onError: (error) => {
    if (error.name === "UnauthorizedError") {
      console.error("Invalid API key");
    } else if (error.name === "NetworkError") {
      console.error("Network error:", error.message);
    } else {
      console.error("Error:", error);
    }
  },
});

try {
  await vision.start();
} catch (error) {
  console.error("Failed to start:", error);
}
```

Result Format

The `onResult` callback receives a `StreamInferenceResult` object:

```
interface StreamInferenceResult {
  id: string; // Result ID
  stream_id: string; // Stream ID
  mode: "clip" | "frame"; // Processing mode used
  model_backend: "overshoot";
  model_name: string; // Model used
  prompt: string; // Task that was run
  result: string; // Model output (always a string - parse JSON if using outputSchema)
  inference_latency_ms: number; // Model inference time
  total_latency_ms: number; // End-to-end latency
  ok: boolean; // Success status
  error: string | null; // Error message if failed
  finish_reason: "stop" | "length" | "content_filter" | null;
}
```

The `finish_reason` field indicates why the model stopped generating:

- `"stop"` — Model finished naturally
- `"length"` — Output was truncated because it hit `maxOutputTokens`. Consider increasing the value or using a longer processing interval.
- `"content_filter"` — Output was blocked by safety filtering

Use Cases

- Real-time text extraction and OCR
- Safety monitoring (PPE detection, hazard identification)
- Accessibility tools (scene description)
- Gesture recognition and control
- Document scanning and alignment detection
- Sports and fitness form analysis
- Video file content analysis
- Screen content monitoring and analysis
- Screen reading accessibility tools
- Tutorial and training content analysis
- Application monitoring and testing

Limits & Billing

- **Concurrent streams:** Maximum 5 streams per API key. Attempting to create a 6th stream returns a 429 error. Close existing streams with `vision.stop()` before starting new ones.
- **Output token rate:** 128 effective tokens per second per stream (see `maxOutputTokens`).
- **Billing:** Streams are billed by duration (stream time), not by number of inference requests. A stream running for 60 seconds costs the same regardless of processing interval. Billing starts at stream creation and ends when the stream is closed or expires.

Error Types

The SDK provides specific error classes for different failure modes:

- `ValidationError` - Invalid configuration or parameters
- `UnauthorizedError` - Invalid or revoked API key
- `NotFoundError` - Stream or resource not found
- `NetworkError` - Network connectivity issues
- `ServerError` - Server-side errors
- `ApiError` - General API errors

Browser Compatibility

Requires browsers with support for:

- WebRTC (RTCPeerConnection)
- MediaStream API
- WebSocket
- Modern JavaScript (ES2020+)
- Screen capture requires getDisplayMedia API (desktop browsers only)

Supported browsers:

- Camera/Video: Chrome 80+, Firefox 75+, Safari 14+, Edge 80+
- Screen capture: Chrome 72+, Firefox 66+, Safari 13+, Edge 79+ (desktop only)

Feedback

As this is an alpha release, we welcome your feedback! Please report issues or suggestions through GitHub issues.

Releases

No releases published

Packages

No packages published

Contributors 3

 **YounesElhjouji** Younes Elhjouji

 **zakariaelh** Zakaria El hjouji

 **claude** Claude

Languages

- **TypeScript** 100.0%