Architecture Decision Document

This document builds collaboratively through step-by-step discovery. Sections are appended as we work through each architectural decision together.

1. Project Context Analysis

1.1 Requirements Overview

Functional Requirements (50 FRs across 10 Capabilities)

The 50 functional requirements organize into **three architectural tiers** based on latency sensitivity and failure-domain coupling:

Tier	Capabilities	FR Count	Latency Sensitivity	Description
Real-Time Audio Path	Audio Streaming (CAP-2), Voice AI Integration (CAP-3)	12	Sub-millisecond	Byte-level audio forwarding between ACS and Voice Live WebSockets. Zero business logic on audio frames.
Call Lifecycle Orchestration	Call Handling (CAP-1), Session Management (CAP-4), Error Handling (CAP-7), Call Transfer (CAP-10)	20	Milliseconds–seconds	State machine transitions, connection setup/teardown, reconnection semantics, graceful degradation.
Outbound Integrations	Tool Call Framework (CAP-5, P0), Security (CAP-6), Monitoring (CAP-8), Configuration (CAP-9)	18	Seconds (tolerable)	HTTP calls to backends via APIM, credential management, metrics emission, config loading.

This tiering drives a critical architectural principle: **code that touches Tier 1 must never import or depend on Tier 2 or Tier 3 modules.** Audio forwarding is a self-contained byte-copying concern.

Non-Functional Requirements (47 NFRs across 6 Categories)

Category	Count	Key Drivers
Performance	10	Audio latency ≤50ms added; call setup ≤2s; 100→1000 concurrent calls
Reliability	8	99.9% uptime; reconnection windows; graceful degradation hierarchy

Category	Count	Key Drivers
Security	10	Zero Trust (18 SEC-REQs); managed identity everywhere; JWT validation on all inbound
Scalability	6	Horizontal pod scaling; 200 WS connections per 100 calls; memory 512MB–1GB/instance
Observability	8	OpenTelemetry traces (5 spans); 16 metrics; 23 structured log events; correlation threading
Maintainability	5	Config-driven prompts (61 params); feature flags; designed for retirement (AP-6)

Critical NFR Tensions:

- **Latency vs. Observability:** Audio path logging must be zero-cost (metrics only, no per-frame logging)
- **Security vs. Latency:** JWT validation on every inbound connection adds ~5ms — acceptable for webhook, concerning for audio path initialization
- **Reliability vs. Complexity:** Reconnection logic with close-code-dependent branching (normal: 3s teardown, abnormal: 5s window) adds significant state machine complexity

1.2 Technical Constraints & Dependencies

#	Constraint	Source	Architectural Impact
TC-1	Java 21 + Spring Boot 4.x + Spring MVC	AP-5 (PRD)	Virtual threads via `spring.threads.virtual.enabled=true`; imperative blocking I/O
TC-2	JSR 356 `@ServerEndpoint` for ACS WebSocket	ACS SDK pattern	Tomcat servlet container required; annotation-based WS handlers
TC-3	`java.net.http.WebSocket` for Voice Live client	Copilot instructions	JDK built-in, no library dependency; requires manual lifecycle management
TC-4	PCM 24000Hz 16-bit mono audio	ACS specification	No transcoding needed (Voice Live same format); ~48KB/s per direction
TC-5	ACS `CallAutomationClient` (synchronous)	SDK design	`azure-communication-callautomation` Maven artifact; blocking calls OK with virtual threads
TC-6	`DefaultAzureCredential` everywhere	Security policy	Managed identity in AKS; `azure-identity` artifact; token cache shared across call instances
TC-7	Maven build system	Ecosystem alignment	ACS Java SDK Maven-native; `./mvnw` wrapper
TC-8	Event Grid direct delivery (no APIM)	Architecture Q&A	~30s ring timeout; APIM latency would cause missed calls

#	Constraint	Source	Architectural Impact
TC-9	ACS callbacks direct (no APIM)	Architecture Q&A	Mid-call webhooks even more latency-sensitive than Event Grid
TC-10	APIM for outbound backend calls only	Architecture Q&A	OAuth2 bearer via managed identity; circuit breaker at APIM layer
TC-11	AKS deployment target	Infrastructure	Horizontal pod autoscaling; 512MB–1GB per instance; container-portable (AP-4)
TC-12	Zero audio storage	SEC-REQ	Audio frames pass through memory only; no disk, no log, no buffer persistence
TC-13	Brazil South region	Deployment	Latency-optimized for Brazilian telecom; ACS + Foundry co-located
TC-14	OpenTelemetry SDK	Observability	`io.opentelemetry` artifacts; auto-instrumentation agent or manual spans
TC-15	**Virtual thread pinning guard**	E-1 (ADR)	All I/O-adjacent locks MUST use `ReentrantLock`, never `synchronized` — `synchronized` pins virtual threads to carrier threads, negating Loom benefits
TC-16	**WebSocket send serialization**	E-3 (ADR)	Outbound WebSocket frames require single-writer pattern (per-connection `ReentrantLock` or queue) — concurrent `sendText`/`sendBinary` calls corrupt frames per RFC 6455
TC-17	**Audio path purity**	E-7 (First Principles)	Audio forwarding path = byte-copy loop with zero business logic. Audio handler and control message handler are distinct classes/methods with no shared code paths. State checks, logging, conditional routing operate only on control messages (JSON events), never on audio frames.

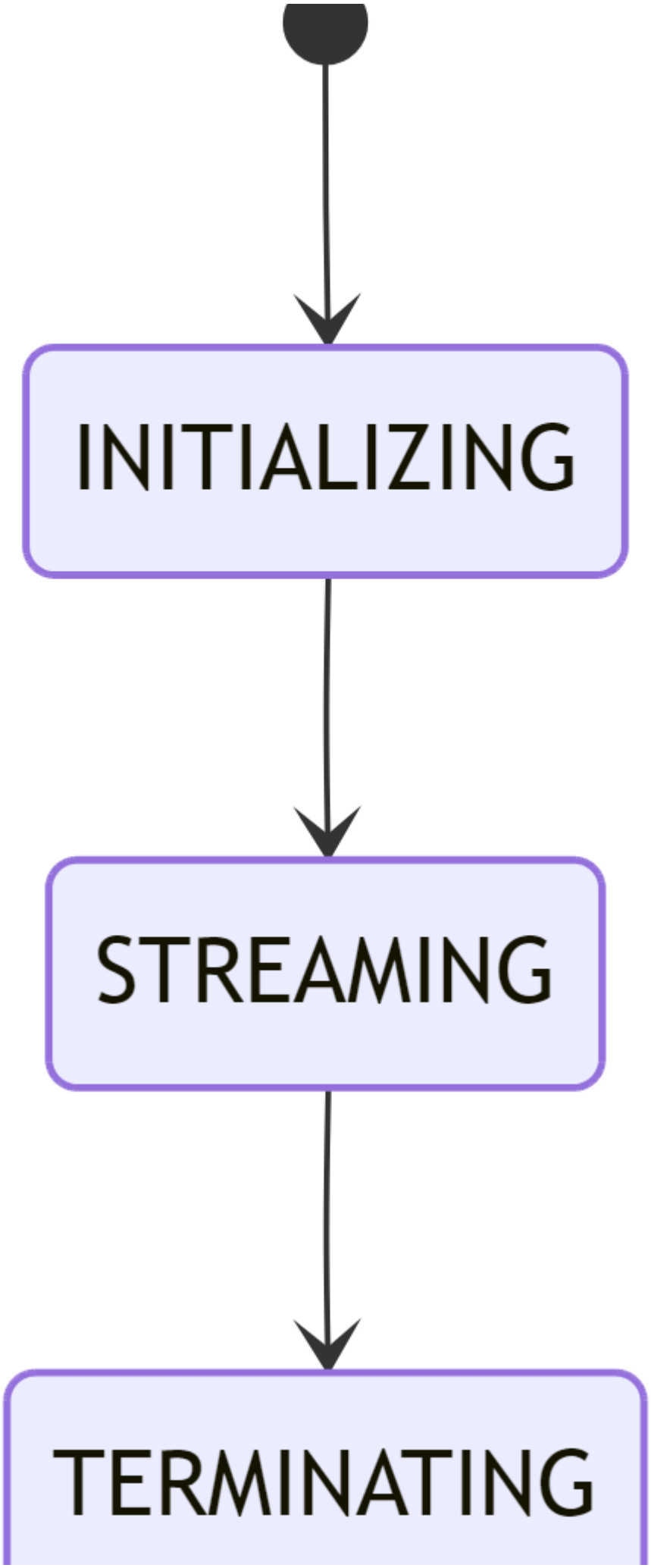
1.3 Cross-Cutting Concerns

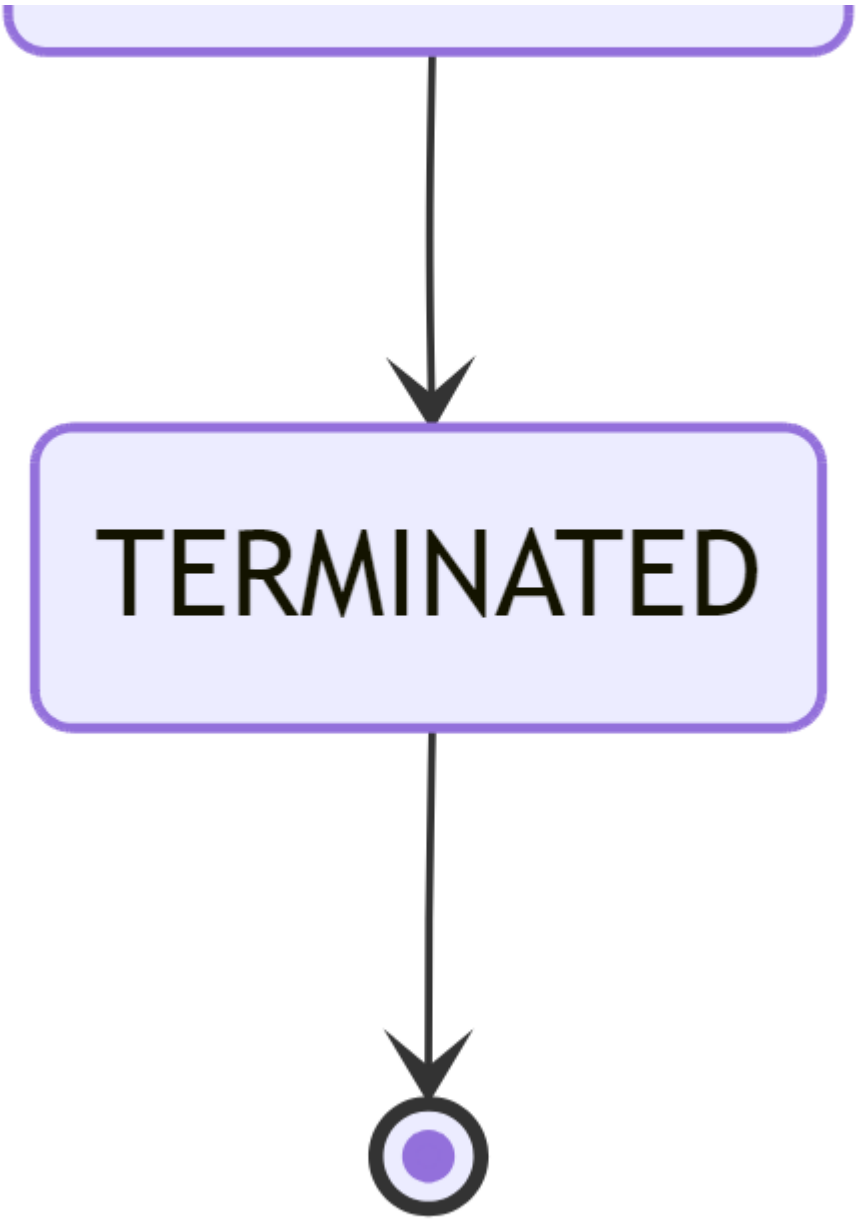
CC-1: Correlation Propagation *(Enhanced by E-6)*

Per-call `CallContext` record holding `correlationId`, OTel span reference, `AtomicReference<CallState>`, and auth state. Passed **explicitly** to all components as a method parameter — ThreadLocal MDC is insufficient because a single call spans two WebSocket threads (ACS inbound + Voice Live outbound) that don't share a thread context. All log events, metrics, and spans receive their correlation ID from `CallContext`, not from thread-local storage.

CC-2: Call State Machine *(Enhanced by E-2, E-10)*

Four behaviorally distinct states:


INITIALIZING
STREAMING
TERMINATING



Source: call-state-machine.mmd

Transitions governed by a **single** `EnumMap<CallState, Set<CallState>>` **transition table** — one point of audit, replaces scattered conditional logic. Invalid transition attempts are logged as errors and rejected (defensive programming).

Tool calls do **not** create additional states. They are tracked as a concurrent count (`AtomicInteger activeToolCalls`) within `CallContext`. Graceful termination (`STREAMING → TERMINATING`) waits for `activeToolCalls == 0` with a bounded timeout (e.g., 5s) before force-closing both WebSockets.

CC-3: Auth Multiplexing *(Enhanced by E-4)*

Three distinct auth flows managed by the service:

1. **Inbound webhook/WS:** JWT validation (ACS-signed tokens)
2. **Outbound to Voice Live:** `DefaultAzureCredential` bearer token
3. **Outbound to APIM/backends:** Managed identity OAuth2 bearer

Token cache is a **system-wide shared resource**. Pre-emptive token refresh at **75% of token lifetime** avoids refresh convoy — lazy on-demand refresh causes thundering herd when all calls discover expired token simultaneously under concurrent load.

CC-4: Graceful Degradation Hierarchy *(Enhanced by E-5, E-8)*

Reconnection semantics differ fundamentally per WebSocket side:

- **ACS WebSocket drops:** Call is **non-recoverable**. ACS will not reconnect to the service — the phone call is over. State machine transitions directly to `TERMINATING`.
- **Voice Live WebSocket drops:** If the service doesn't support session resumption, "reconnection" = **new session + UX recovery** (play apology prompt, restart conversation context). This is architecturally distinct from transparent reconnection. The PRD's "5s reconnection window" applies here — it's the budget for establishing a new Voice Live session before giving up and terminating.

Health probe state machine: Readiness probe includes credential staleness grace period (token lifetime + 60s buffer) and DNS cache TTL awareness. Require **N consecutive failures** before marking unhealthy to prevent false-negative readiness during transient Azure AD latency.

Degradation ordering: tool call failure (graceful) → Voice Live reconnection attempt → apology prompt → call termination.

CC-5: Configuration Validation

61 configuration parameters validated at startup. Invalid configuration prevents service start (fail-fast). Runtime configuration changes (feature flags, prompt updates) validated before application. Categories: connection endpoints, timeouts, retry policies, audio parameters, security settings, observability thresholds, prompt templates.

CC-6: Resource Cleanup Determinism

Each call holds two WebSocket connections, an HTTP client reference (for tool calls), OTel spans, and auth tokens. Cleanup must be **deterministic and ordered:**

1. Drain active tool calls (bounded wait)
2. Close Voice Live WSS (send close frame)
3. Close ACS WSS (send close frame)
4. End OTel span
5. Release `CallContext`

Abnormal termination (crash, OOM) must not leak resources. `try-finally` blocks with defensive close checks. WebSocket `onClose`/`onError` handlers trigger cleanup from either side.

CC-7: Correlated Failure Isolation *(E-9)*

Sharp distinction between **per-call isolated resources** (safe to fail individually) and **system-wide shared resources** (correlated failure risk):

Shared Resource	Failure Mode	Degradation Strategy
Token cache (`DefaultAzureCredential`)	Refresh fails	Fall back to per-call token acquisition (slower, higher AAD load, but functional)
HTTP client pool	Pool exhaustion	Tool calls fail with bounded timeout; audio path unaffected; emit metric
Health state	False-negative readiness	Require N consecutive failures before marking unhealthy (circuit-breaker on health transitions)
DNS cache	Stale resolution	Bounded TTL with async refresh; don't block audio path on DNS

Per-call failures affect one customer. Shared resource failures affect **all concurrent calls** — these require explicit circuit-breaker patterns and fallback strategies.

1.4 Architectural Principles (from PRD)

ID	Principle	Summary
AP-1	Protocol Translator	Service is an audio bridge, not a processing engine
AP-2	Two-WebSocket Model	One inbound (ACS), one outbound (Voice Live) per call
AP-3	Thin API Gateway	APIM for outbound only; inbound path is direct
AP-4	Container-Portable	No cloud-specific runtime dependencies beyond config
AP-5	Virtual Threads + Spring MVC	Imperative blocking I/O via Project Loom
AP-6	Designed for Retirement	Replaceable when ACS natively integrates Voice Live

1.5 Elicitation Insights Summary

Methods Executed: Architecture Decision Records, First Principles Analysis

Key Insights:

- Virtual thread pinning is a real operational risk requiring coding standards enforcement (E-1)
- WebSocket concurrency requires explicit serialization — not handled by the JDK API automatically (E-3)
- Audio path and control message path must be physically separated in code (E-7)
- Reconnection is semantically different per WebSocket side (E-8)
- The state machine should be minimal (4 states) with tool calls as concurrent activity, not states (E-10)
- Correlated failures (token expiry, pool exhaustion) are the primary reliability threat, not per-call bugs (E-9)

---

2. Starter Template Evaluation

2.1 Primary Technology Domain

Java backend service / API — no frontend, no CLI, no mobile. A Spring Boot application serving REST endpoints (Event Grid webhooks, ACS callbacks) and a WebSocket server endpoint (ACS audio streaming), with outbound WebSocket and HTTP clients.

2.2 Starter Options Considered

Option	Description	Verdict
A: Spring Initializr	Canonical Spring Boot starter via `start.spring.io`. Clean, current, production-ready.	☑ Base
B: ACS + OpenAI Java Sample	Azure-Samples reference. Has ACS `@ServerEndpoint` patterns but targets older Java, OpenAI (not Voice Live), sample-quality code.	Reference only
C: Initializr + ACS Sample as Reference	Create from Initializr with exact deps, cherry-pick ACS WebSocket patterns from the sample without forking.	☑ **Selected**

2.3 Selected Starter: Spring Initializr + ACS Sample as Reference

Rationale:

- Clean, current dependency tree from day one
- Production-ready Spring Boot configuration
- Cherry-pick only the ACS patterns we need (JSR 356 endpoint, `StreamingData` parsing) without inheriting sample-quality compromises
- Virtual threads enabled from the start

Initialization Command:

```
curl https://start.spring.io/starter.zip \
  -d type=maven-project \
  -d language=java \
  -d javaVersion=21 \
  -d bootVersion=4.0.0 \
  -d groupId=com.sofia \
  -d artifactId=ivr-bridge \
  -d name=sofia-ivr-bridge \
  -d description="ACS Voice Live IVR Bridge Service" \
  -d packageName=com.sofia.ivr \
  -d dependencies=web,websocket,actuator,validation,configuration-processor \
  -o ivr-bridge.zip
```

Note: Adjust `bootVersion` to the latest Spring Boot 4.0.x GA available at project start time.

2.4 Dependencies

Direct Dependencies

Artifact	Source	Purpose
`spring-boot-starter-web`	Spring Boot BOM	Embedded Tomcat, Spring MVC, Jackson
`spring-boot-starter-websocket`	Spring Boot BOM	Spring WebSocket support
`spring-boot-starter-actuator`	Spring Boot BOM	Health endpoints (`/actuator/health`), Micrometer metrics
`spring-boot-starter-validation`	Spring Boot BOM	Bean validation (Hibernate Validator)
`spring-boot-configuration-processor`	Spring Boot BOM	`@ConfigurationProperties` metadata generation
`azure-communication-callautomation`	Azure SDK BOM	ACS Call Automation — answer/hangup, `StreamingData` parsing
`azure-identity`	Azure SDK BOM	`DefaultAzureCredential` for managed identity
`azure-sdk-bom`	Maven Central	Version management for all `com.azure` artifacts
`opentelemetry-spring-boot-starter`	OTel	OpenTelemetry autoconfig — traces, metrics, log bridge
`resilience4j-spring-boot3`	Resilience4j	Circuit breaker for tool call dispatch (Decision 6)
`logstash-logback-encoder`	Maven Central	JSON structured logging for Logback (Decision 10)
`spring-boot-starter-test`	Spring Boot BOM (test)	JUnit 5, Mockito, AssertJ

BOM Ordering (Critical)

```
<!-- pom.xml — correct BOM ordering -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>4.0.x</version>  <!-- TOP PRIORITY — wins all version conflicts -->
</parent>

<dependencyManagement>
  <dependencies>
    <!-- Azure SDK BOM — SECOND PRIORITY -->
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-sdk-bom</artifactId>
      <version>${azure-sdk-bom.version}</version>
```

```
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!-- OTel versions managed by spring-boot-starter, no separate BOM needed -->
  </dependencies>
</dependencyManagement>
```

Spring Boot parent POM takes highest precedence. Maven's "nearest-wins" ensures Spring Boot's Jackson and Micrometer versions win over Azure SDK's.

ADR-7: Azure SDK HTTP Transport — Keep Netty Default

Status: Accepted | **Date:** 2026-02-15

Context: Azure SDK uses `azure-core-http-netty` as its default HTTP transport for outbound calls (token acquisition, ACS REST API). Our app uses embedded Tomcat for inbound. This puts two HTTP stacks on the classpath.

Decision: Keep `azure-core-http-netty` (default). Do not swap to `azure-core-http-jdk-httpclient`.

Rationale:

1. No functional conflict — Tomcat handles inbound, Netty handles Azure SDK outbound internally. Different roles, no overlap.
2. Most tested path — vast majority of Azure SDK Java users run Netty transport.
3. Low overhead — ~15 JARs / ~4MB. Negligible at 512MB–1GB container sizing.
4. Virtual thread coexistence is safe — Netty's `EventLoopGroup` threads handle low-frequency SDK I/O. Virtual threads handle WebSocket connections.
5. Reversible — swap to JDK HttpClient is a single POM exclusion + dependency addition.

Escape hatch: If profiling reveals Netty overhead, swap to `azure-core-http-jdk-httpclient` with one POM change.

2.5 Starter Setup Steps (P-1)

Three setup items **not provided** by Spring Initializr that must be configured in the first implementation story:

2.5.1 `ServerEndpointExporter` Bean Registration

Spring Boot does **not** auto-register JSR 356 `@ServerEndpoint` classes. Without this bean, Tomcat silently ignores WebSocket endpoints — no error message, just non-registration.

```java
@Configuration
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

2.5.2 Package Layout Convention

```
com.sofia.ivr
├── config/          # Spring configuration, WebSocket config, properties binding
├── webhook/         # REST controller for Event Grid + ACS callbacks
├── websocket/       # JSR 356 @ServerEndpoint (ACS audio) + Voice Live WS client
├── bridge/          # Audio bridge service, audio forwarding logic
├── toolcall/        # Tool call dispatcher, HTTP client to APIM
├── model/           # CallContext, CallState enum, DTOs, Voice Live event types
└── health/          # Custom health indicators, readiness probe logic
```

2.5.3 Default Configuration (`application.yaml`)

```
spring:
  threads:
    virtual:
      enabled: true    # Virtual threads on Tomcat executor — MUST be present from
day one
  application:
    name: sofia-ivr-bridge
```

2.6 Reference Patterns from ACS Sample

Cherry-picked from ACS CallAutomation OpenAI Java Sample — **reference only, not forked:**

- JSR 356 `@ServerEndpoint` audio streaming handler pattern
- `StreamingData.parse()` for inbound audio frame parsing
- `OutStreamingData.getStreamingDataForOutbound()` for outbound audio frame formatting
- `CallAutomationClient` answer/hangup wiring

2.7 Architectural Decisions Established by Starter

Decision	Choice
Project structure	Standard Maven `src/main/java`, `src/main/resources`, `src/test/java`
Runtime	Embedded Tomcat (JSR 356 + Spring MVC + virtual threads)
Configuration	`application.yaml` with Spring profiles
Health	Actuator `/actuator/health` (readiness/liveness)
Metrics	Micrometer → OTel exporter
Testing	JUnit 5 + Spring Boot Test
Build	Maven with `./mvnw` wrapper

3. Core Architectural Decisions

3.1 Decision 1 — In-Memory Call State Store

Recommendation: `ConcurrentHashMap<String, CallContext>` for per-call state.

Rationale: Each active call is pinned to exactly one pod instance via persistent WebSocket connections (ACS→Service and Service→VoiceLive). The `CallContext` object (call ID, session state, tool call queue, audio state) is accessed exclusively by the virtual threads handling that call's WebSocket messages. No cross-pod sharing is needed or beneficial.

Accepted: ConcurrentHashMap with TTL-based eviction via a scheduled virtual-thread task.

ADR-8: Spring Boot 4.0.x Baseline

Status: Accepted | **Date:** 2026-02-15

Context: Spring Boot 3.4.x is the current stable LTS line. Spring Boot 4.0.x (based on Spring Framework 7) brings first-class virtual thread support, improved AOT compilation, and Jakarta EE 11 alignment. The PRD specifies Spring Boot 4.x explicitly. Project timeline targets production in H2 2026, by which time 4.0.x will have multiple patch releases.

Decision: Use Spring Boot 4.0.x as the baseline.

Rationale:

1. PRD mandates 4.x — overriding to 3.4.x would contradict stakeholder requirements.
2. Virtual thread integration is richer — Tomcat executor, `@Async`, and `@Scheduled` all virtual-thread-aware by default.
3. Jakarta EE 11 alignment avoids future namespace migration.
4. Risk is bounded — this is a greenfield service (no legacy migration), and 4.0.x will be production-ready by deployment target.

Risk: Early 4.0.x patch versions may have edge-case bugs. Mitigated by: pinning to a specific patch (e.g., 4.0.2), comprehensive test suite, and the fact that the project's dependency surface is narrow (Web, WebSocket, Actuator, Azure SDK).

ADR-9: OpenTelemetry Spring Boot Autoconfig over Java Agent

Status: Accepted | **Date:** 2026-02-15

Context: Two approaches exist for OTel instrumentation in Spring Boot: (A) the OTel Java Agent (`-javaagent:opentelemetry-javaagent.jar`) which auto-instruments bytecode at runtime, or (B) the OTel Spring Boot Starter (`opentelemetry-spring-boot-starter`) which uses Spring's Micrometer integration + OTel exporter. Both produce OTel-compatible traces and metrics.

Decision: Use `opentelemetry-spring-boot-starter` (autoconfig) instead of the Java Agent.

Rationale:

1. Debuggability — no bytecode manipulation means standard breakpoints work predictably in IDE.

2. Spring Boot 4.x native — Micrometer integration is first-class; the autoconfig starter aligns with Spring's intended telemetry model.
3. Explicit dependency — visible in POM, version-managed by BOM, no external JAR to manage in Dockerfile.
4. Sufficient coverage — Spring MVC, WebClient, JDBC, and Actuator are all auto-instrumented via Micrometer. Custom spans for WebSocket paths use `@Observed` annotation or manual `Tracer` API.
5. Simpler container — no `JAVA_TOOL_OPTIONS=-javaagent:...` environment variable, no volume mount for the agent JAR.

Trade-off: Java Agent auto-instruments more libraries (e.g., Netty internals, raw JDBC drivers) without code changes. If deep Azure SDK internal tracing is needed later, the agent can be added alongside the starter without conflict.

ADR-10: ConcurrentHashMap over Redis for Call State

Status: Accepted | **Date:** 2026-02-15

Context: Each active call creates two WebSocket connections (ACS→Service, Service→VoiceLive) pinned to the same pod instance for the call's lifetime. Call state (`CallContext`) includes call metadata, session state, tool call queue, and audio state. The question: should this state live in-process (`ConcurrentHashMap`) or in an external store (Redis)?

Decision: Use `ConcurrentHashMap<String, CallContext>` for call state, with TTL-based eviction.

Rationale:

1. **No sharing need** — WebSocket pinning means only one pod ever accesses a call's state. Redis adds zero value for cross-pod lookup.
2. **Latency** — `ConcurrentHashMap.get()` is ~50ns. Redis round-trip is ~500µs–2ms. For an audio bridge processing frames every 20ms, this 10,000–40,000x penalty is unacceptable in the hot path.
3. **Failure domain** — Redis introduces a new dependency that can fail independently. A Redis outage would kill all calls, not just the affected pod's.
4. **Operational cost** — Redis requires provisioning, monitoring, patching, and network security rules. `ConcurrentHashMap` requires nothing.
5. **Scalability model** — Horizontal scaling adds pods, each with its own map. No shared-state bottleneck.

Eviction strategy: A scheduled virtual-thread task runs every 60s, removing entries where `CallContext.lastActivity` exceeds the configured TTL (default: 30 minutes). This handles orphaned entries from ungraceful disconnects.

Escape hatch: If a future requirement demands call state survive pod restarts (e.g., call transfer across pods), Redis can be introduced for that specific use case without rearchitecting — the `CallStateStore` interface can be extracted and given a Redis implementation.

ADR-11: Nimbus JOSE+JWT for All Inbound JWT Validation (No Spring Security)

Status: Accepted | **Date:** 2026-02-15

Problem Statement

This service receives inbound requests from three distinct sources, each authenticated via JWT but with different issuers, key sources, and transport mechanisms. Additionally, the service makes outbound authenticated calls to backend APIs via APIM. The question: which library/framework should handle JWT validation for inbound paths, and does outbound token acquisition influence this choice?

Inbound Authentication Paths

Path 1 — Event Grid Webhook Delivery → REST Controller (`/api/events`)

- Token type: Entra ID bearer token in `Authorization` header
- Issuer: `https://login.microsoftonline.com/{tenantId}/v2.0`
- Audience: Application ID URI (`api://<client-id>`)
- JWKS source: `https://login.microsoftonline.com/{tenantId}/discovery/v2.0/keys`
- OIDC-compliant: Yes — publishes `/.well-known/openid-configuration`
- Volume: One POST per incoming call (low frequency)

Path 2 — ACS Call Automation Callbacks → REST Controller (`/api/callbacks/{contextId}`)

- Token type: ACS-signed JWT in `Authorization` header
- Issuer: `https://acscallautomation.communication.azure.com`
- Audience: Callback base URI (`https://sofia-ivr.example.com/api/callbacks`)
- JWKS source: `https://acscallautomation.communication.azure.com/keys`
- OIDC-compliant: **No** — no `/.well-known/openid-configuration` endpoint
- Volume: Multiple callbacks per call (CallConnected, PlayCompleted, etc.)

Path 3 — ACS Audio WebSocket → JSR 356 `@ServerEndpoint` (`wss://host/ws/audio`)

- Token type: ACS JWT in WebSocket upgrade request (query parameter or header)
- Issuer: Same as Path 2 (ACS)
- Validation point: `ServerEndpointConfig.Configurator.modifyHandshake()` — runs during HTTP→WSS upgrade, before `@OnOpen`
- Transport: JSR 356 (Tomcat-managed), **outside Spring's `DispatcherServlet` and filter chain**
- Volume: Once per call (connection establishment)

Outbound Authentication (Not Part of This Decision)

The service calls APIM-fronted backends (Oracle BRM, CRM) using `DefaultAzureCredential` with managed identity. This is Azure SDK territory — token acquisition, caching, and refresh are handled by `com.azure.identity`. Spring Security's OAuth2 Client (`OAuth2AuthorizedClientManager`, `client_credentials` grant) could theoretically replace this, but:

1. `DefaultAzureCredential` is already required for ACS SDK and Voice Live API authentication
2. Having two competing token acquisition mechanisms adds confusion without benefit
3. Managed identity token acquisition is not an OAuth2 client_credentials flow — it's a metadata endpoint call

Conclusion: Outbound auth is handled by Azure SDK regardless. This decision concerns inbound validation only.

Options Evaluated

Option A: `spring-security-oauth2-resource-server`

How it works: Spring Security filter chain intercepts HTTP requests and validates JWTs against a configured `JwtDecoder`. Auto-configured via `spring.security.oauth2.resourceserver.jwt.issuer-uri` property for OIDC-compliant issuers.

Coverage by path:

- Path 1 (Event Grid / Entra ID): ☑ Native fit — properties-only config, zero Java code for single-issuer OIDC. Auto-discovers JWKS URI, validates issuer/expiry/signature.
- Path 2 (ACS Callbacks): ⚠ Partial — ACS is not OIDC-compliant (no discovery endpoint). Multi-issuer requires `JwtIssuerAuthenticationManagerResolver`, which does OIDC discovery per issuer. ACS would fail discovery. Workaround: register a custom `AuthenticationManager` for the ACS issuer that uses Nimbus internally — ~40 lines of framework plumbing to do what is essentially a manual Nimbus call.
- Path 3 (ACS WebSocket): ❌ Not applicable — JSR 356 `@ServerEndpoint` is managed by Tomcat's WebSocket subsystem, not Spring's `DispatcherServlet`. The Security filter chain never intercepts WebSocket upgrade requests for JSR 356 endpoints. Validation must be done manually in `ServerEndpointConfig.Configurator.modifyHandshake()` using Nimbus directly.

Dependencies added: `spring-security-core`, `spring-security-config`, `spring-security-web`, `spring-security-oauth2-core`, `spring-security-oauth2-jose` (5 JARs).

Effective coverage: **1 of 3 paths** through the framework. Remaining 2 paths use manual Nimbus.

Option A variant: Properties-Only Auto-Config (Spring Boot magic)

Spring Boot allows zero-code JWT validation via:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://login.microsoftonline.com/{tenantId}/v2.0
```

Limitations:

1. Single `issuer-uri` — no array syntax for multiple issuers
2. `issuer-uri` triggers OIDC discovery — fails for ACS
3. `jwk-set-uri` alternative skips discovery but still single-issuer and loses auto-issuer-validation
4. WebSocket path remains uncovered regardless

This variant makes the framework even more attractive for single-issuer REST services, but does not change the fundamental multi-issuer + WebSocket gap.

Option B: `nimbus-jose-jwt` directly

How it works: Application code creates `DefaultJWTProcessor` instances configured with issuer-specific `JWSKeySelector` and `JWTClaimsVerifier`. Validation is a direct method call — `processor.process(token, null)`.

Coverage by path:

- Path 1 (Event Grid): ☑ `JwtValidator` bean with Entra ID issuer/JWKS/audience config
- Path 2 (ACS Callbacks): ☑ `JwtValidator` bean with ACS issuer/JWKS/audience config
- Path 3 (ACS WebSocket): ☑ Same `JwtValidator` (ACS) called from `Configurator.modifyHandshake()`

Dependencies added: **None new** — `nimbus-jose-jwt`, `json-smart`, and `accessors-smart` are already transitive dependencies of Spring Boot (used internally by Spring's OAuth2 support even when Spring Security is not on classpath).

Effective coverage: **3 of 3 paths** with one consistent pattern.

Implementation cost: ~60 lines total — `JwtValidatorFactory` (~30 lines) + `JwtValidator` (~30 lines). Each path consumes it as a one-line `validator.validate(authHeader)` call.

Option C: `azure-spring-boot-starter-active-directory`

How it works: Azure-specific Spring Boot starter that auto-configures Entra ID as the OAuth2 resource server.

Coverage by path:

- Path 1 (Event Grid): ☑ Auto-configured for Entra ID
- Path 2 (ACS Callbacks): ❌ Designed for Entra ID only, no multi-issuer support
- Path 3 (ACS WebSocket): ❌ Same JSR 356 limitation as Option A

Additional concerns: Opinionated — pulls in Spring Security as a transitive dependency. Designed for services with user-facing auth flows (scopes, roles, groups), not machine-to-machine webhook validation.

Eliminated early — strictly worse than Option A for this use case.

Option D: Hybrid — Spring Security for Event Grid + Nimbus for ACS

Technically viable:

```
.authorizeHttpRequests(auth -> auth
    .requestMatchers("/api/events").authenticated()       // Spring Security
    .requestMatchers("/api/callbacks/**").permitAll()     // manual Nimbus
    .requestMatchers("/actuator/health/**").permitAll()
)
```

Concerns:

- Two validation mental models in one service ("which paths use which mechanism?")
- Spring Security filter chain runs on every HTTP request, including `permitAll()` callbacks — evaluated but permitted, adding overhead to the latency-sensitive callback path
- 5 Spring Security JARs added for one endpoint

- New team members must understand the split

Eliminated — complexity/benefit ratio too high for protecting one low-volume endpoint.

Decision

Use `nimbus-jose-jwt` directly for all three inbound JWT validation paths. Do not add Spring Security to the dependency tree.

Architecture:

- `JwtValidatorFactory` — creates configured `JwtValidator` instances per issuer (Entra ID, ACS)
- `JwtValidator` — wraps `DefaultJWTProcessor<SecurityContext>` with issuer-specific key selector and claims verifier
- `RemoteJWKSet` (Nimbus built-in) — handles JWKS fetching, caching (5-minute default), and automatic key rotation
- Two `JwtValidator` beans wired via `@Configuration`:
  - `eventGridValidator` — Entra ID issuer, Entra ID JWKS URI, application audience
  - `acsValidator` — ACS issuer, ACS JWKS URI, callback audience
- Outbound token acquisition remains `DefaultAzureCredential` (Azure SDK) — completely separate concern

Rationale

1. **Uniform pattern** — `validator.validate(authHeader)` is the same call in REST controllers and WebSocket configurator. One pattern to learn, test, and debug.
2. **Zero new dependencies** — Nimbus JARs already on classpath as Spring Boot transitive dependencies.
3. **Multi-issuer by design** — two `JwtValidator` beans, each configured for its issuer. Adding a third issuer later is one more `factory.create()` call.
4. **Non-OIDC issuer support** — ACS doesn't publish OIDC discovery. Nimbus's `RemoteJWKSet` points directly at a JWKS URI — no discovery needed.
5. **WebSocket-native** — JSR 356 `Configurator.modifyHandshake()` is a plain Java method call. Nimbus works perfectly here. Spring Security's filter chain is structurally incapable of intercepting this path.
6. **No filter chain overhead** — no Spring Security filters evaluate on the latency-sensitive callback path.
7. **Simpler mental model** — no "which endpoints are behind Spring Security and which aren't?" question.
8. **Machine-to-machine service** — this service has no user sessions, no login flows, no role-based authorization, no `@PreAuthorize` annotations. Spring Security's rich authorization infrastructure provides zero value here.

Trade-offs Accepted

- **No Spring Security audit events** — `AuthenticationSuccessEvent`, `AuthorizationDeniedEvent` etc. are not emitted. Mitigated: application-level logging in `JwtValidator` on success/failure, exported via OTel.
- **Manual JWKS cache management** — Nimbus's `RemoteJWKSet` handles this automatically, but there's no Spring-managed lifecycle. Mitigated: `RemoteJWKSet` is thread-safe and handles its own cache TTL internally.

- **~60 lines of custom code** — vs zero lines for properties-only Spring Security on Path 1. Accepted: those 60 lines replace what would be ~40 lines of Spring Security ACS workaround code + the hybrid complexity.

Escape Hatch

If future requirements introduce user-facing endpoints with role-based authorization (e.g., admin dashboard, API for human operators), Spring Security can be added at that point. The `JwtValidator` classes would coexist — Spring Security covers the new user-facing paths, `JwtValidator` continues to cover the existing machine-to-machine paths. No rearchitecting needed.

Reference Implementation

Core Infrastructure — `JwtValidatorFactory`

Reusable factory that creates configured validators per issuer:

```
@Component
public class JwtValidatorFactory {

    private final Map<String, JWSKeySelector<SecurityContext>> keySelectors = new
ConcurrentHashMap<>();

    /**
     * Create a validator for a specific issuer + JWKS endpoint + audience.
     */
    public JwtValidator create(String issuer, String jwksUri, String
expectedAudience) {
        JWSKeySelector<SecurityContext> keySelector =
keySelectors.computeIfAbsent(jwksUri, uri -> {
            try {
                JWKSource<SecurityContext> jwkSource = new RemoteJWKSet<>(new
URL(uri),
                    new DefaultResourceRetriever(5000, 5000));  // connect/read
timeout
                return new JWSVerificationKeySelector<>(JWSAlgorithm.RS256,
jwkSource);
            } catch (MalformedURLException e) {
                throw new IllegalArgumentException("Invalid JWKS URI: " + uri, e);
            }
        });

        DefaultJWTClaimsVerifier<SecurityContext> claimsVerifier = new
DefaultJWTClaimsVerifier<>(
            new JWTClaimsSet.Builder()
                .issuer(issuer)
                .audience(expectedAudience)
                .build(),
            Set.of("sub", "iat", "exp")  // required claims
        );
```

```
        return new JwtValidator(keySelector, claimsVerifier);
    }
}
```

Core Infrastructure — `JwtValidator`

Wraps Nimbus `DefaultJWTProcessor` for a single issuer:

```
public class JwtValidator {

    private final ConfigurableJWTProcessor<SecurityContext> processor;

    public JwtValidator(JWSKeySelector<SecurityContext> keySelector,
                        JWTClaimsVerifier<SecurityContext> claimsVerifier) {
        this.processor = new DefaultJWTProcessor<>();
        this.processor.setJWSKeySelector(keySelector);
        this.processor.setJWTClaimsVerifier(claimsVerifier);
    }

    public JWTClaimsSet validate(String bearerToken) {
        String token = bearerToken.startsWith("Bearer ")
            ? bearerToken.substring(7)
            : bearerToken;
        try {
            return processor.process(token, null);
        } catch (ParseException | BadJOSEException | JOSEException e) {
            throw new UnauthorizedException("Invalid JWT: " + e.getMessage(), e);
        }
    }
}
```

Configuration — Wire Both Issuers

```
@Configuration
public class AuthConfig {

    @Bean
    public JwtValidator eventGridValidator(JwtValidatorFactory factory,
                                           AuthProperties props) {
        return factory.create(
            "https://login.microsoftonline.com/" + props.tenantId() + "/v2.0",
            "https://login.microsoftonline.com/" + props.tenantId() +
"/discovery/v2.0/keys",
            props.eventGridAudience()
        );
    }

    @Bean
    public JwtValidator acsValidator(JwtValidatorFactory factory,
```

```
                                        AuthProperties props) {
        return factory.create(
            "https://acscallautomation.communication.azure.com",
            "https://acscallautomation.communication.azure.com/keys",
            props.acsCallbackAudience()
        );
    }
}
```

Path 1 — Event Grid Webhook (Entra ID token)

```
@RestController
@RequestMapping("/api/events")
public class EventGridController {

    private final JwtValidator eventGridValidator;

    @PostMapping
    public ResponseEntity<?> handleEvent(
            @RequestHeader("Authorization") String authHeader,
            @RequestBody String body) {
        eventGridValidator.validate(authHeader);
        // ... process Event Grid events (IncomingCall, SubscriptionValidation)
    }
}
```

Path 2 — ACS Callbacks (ACS-signed token)

```
@RestController
@RequestMapping("/api/callbacks")
public class CallbackController {

    private final JwtValidator acsValidator;

    @PostMapping("/{contextId}")
    public ResponseEntity<Void> handleCallback(
            @PathVariable String contextId,
            @RequestHeader("Authorization") String authHeader,
            @RequestBody String body) {
        acsValidator.validate(authHeader);
        // ... process ACS callback (CallConnected, CallDisconnected, etc.)
    }
}
```

Path 3 — ACS WebSocket Upgrade (ACS token in upgrade request)

```
public class AcsWebSocketConfigurator extends ServerEndpointConfig.Configurator {

    // JSR 356 Configurator is instantiated by Tomcat, not Spring —
    // static reference set from Spring context at startup
    private static JwtValidator acsValidator;

    public static void init(JwtValidator validator) {
        acsValidator = validator;
    }

    @Override
    public void modifyHandshake(ServerEndpointConfig sec,
                                HandshakeRequest request,
                                HandshakeResponse response) {
        String token = extractTokenFromQuery(request);
        JWTClaimsSet claims = acsValidator.validate(token);
        // Store validated claims in user properties for @OnOpen access
        sec.getUserProperties().put("claims", claims);
    }

    private String extractTokenFromQuery(HandshakeRequest request) {
        // ACS sends token as query parameter on WebSocket upgrade
        Map<String, List<String>> params = request.getParameterMap();
        List<String> tokens = params.get("access_token");
        if (tokens == null || tokens.isEmpty()) {
            throw new UnauthorizedException("Missing access_token in WebSocket
upgrade");
        }
        return tokens.get(0);
    }
}
```

Configurator Initialization at Startup

```
@Component
public class WebSocketAuthInitializer {

    public WebSocketAuthInitializer(@Qualifier("acsValidator") JwtValidator
acsValidator) {
        AcsWebSocketConfigurator.init(acsValidator);
    }
}
```

All three paths use the same `validator.validate()` call — one pattern, two issuer-specific beans, zero framework magic.

3.2 Decision 3 — Event Grid SubscriptionValidation Handling

Recommendation: Synchronous handshake (Mechanism A).

When an Event Grid subscription is created, Event Grid sends a `SubscriptionValidationEvent` to prove endpoint ownership. The service must respond synchronously with the `validationCode` in the response body.

Flow:

1. Event Grid POSTs to `/api/events` with `eventType: Microsoft.EventGrid.SubscriptionValidationEvent`
2. Body contains `data.validationCode`
3. Service responds `200 OK` with `{ "validationResponse": "<validationCode>" }`
4. Subscription becomes active

Key details:

- Validation events carry Entra ID JWT when AAD auth is configured — `eventGridValidator.validate()` applies normally, no bypass needed
- Handled inline in the same `EventGridController` — no separate endpoint
- Deployment ordering constraint: service must be running and JWT-configured **before** Terraform creates the Event Grid subscription (chicken-and-egg)
- Asynchronous validation URL (Mechanism B) exists as fallback but is unnecessary when we control the endpoint

Accepted: Synchronous handshake, inline in `EventGridController`, JWT validated on all events including validation.

3.3 Decision 4 — Voice Live Event Schema Modeling

Recommendation: Sealed interfaces + records (Java 21 pattern matching).

Voice Live API uses the OpenAI Realtime API-compatible event schema. Events flow bidirectionally over WebSocket.

Inbound events (Voice Live → Service):

Event Type	Purpose
`session.created`	Session confirmed
`session.updated`	Session config acknowledged
`input_audio_buffer.speech_started`	VAD detected speech
`input_audio_buffer.speech_stopped`	VAD detected silence
`response.audio.delta`	Audio chunk to play to caller
`response.audio.done`	Audio stream complete
`response.function_call_arguments.delta`	Streaming tool call args
`response.function_call_arguments.done`	Tool call ready to execute
`response.done`	Full response complete

Event Type	Purpose
`error`	Error from Voice Live

Outbound events (Service → Voice Live):

Event Type	Purpose
`session.update`	Configure session (system prompt, tools, voice)
`input_audio_buffer.append`	Forward caller audio
`input_audio_buffer.clear`	Clear audio buffer
`conversation.item.create`	Inject tool call result
`response.create`	Trigger response generation

Modeling approach:

```
public sealed interface VoiceLiveEvent permits
    SessionCreated, SessionUpdated, ResponseAudioDelta,
    ResponseAudioDone, FunctionCallArgumentsDelta,
    FunctionCallArgumentsDone, ResponseDone, ErrorEvent,
    SpeechStarted, SpeechStopped {
    String type();
}

public record ResponseAudioDelta(String type, String eventId,
                                 String responseId, byte[] delta)
    implements VoiceLiveEvent {}

public record FunctionCallArgumentsDone(String type, String callId,
                                        String name, String arguments)
    implements VoiceLiveEvent {}
```

Dispatch via pattern matching:

```
switch (event) {
    case ResponseAudioDelta delta -> bridge.forwardAudio(delta.delta());
    case FunctionCallArgumentsDone tool -> toolHandler.dispatch(tool);
    case ErrorEvent error -> handleError(error);
    // compiler warns on missing cases
}
```

Outbound commands modeled as a separate sealed interface (`VoiceLiveCommand`).

Deserialization: Custom Jackson `StdDeserializer` reads the `"type"` field and delegates to the correct record class. One-time setup cost.

Options rejected:

- Single `VoiceLiveEvent` class + `JsonNode data` — no compile-time exhaustiveness, runtime `dataAs()` failures
- Jackson `@JsonTypeInfo` polymorphic — annotation-heavy, `@JsonSubTypes` maintenance burden, abstract class prevents records

Accepted: Sealed interfaces + records for both inbound events and outbound commands.

3.4 Decision 5 — Tool Call HTTP Dispatch Pattern

Recommendation: `RestClient` (Spring Boot 4.x synchronous HTTP client) + Resilience4j circuit breaker.

When Voice Live sends `response.function_call_arguments.done`, the service calls a backend API via APIM. This happens mid-conversation — the caller hears silence while waiting.

Implementation:

```
@Component
public class ToolCallDispatcher {

    private final RestClient restClient;
    private final Map<String, ToolDefinition> toolRegistry;
    private final CircuitBreaker circuitBreaker;

    public String dispatch(FunctionCallArgumentsDone toolCall) {
        ToolDefinition def = toolRegistry.get(toolCall.name());
        if (def == null) throw new UnknownToolException(toolCall.name());

        return Decorators.ofSupplier(() ->
                restClient.post()
                    .uri(def.apimEndpoint())
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(toolCall.arguments())
                    .retrieve()
                    .body(String.class))
            .withCircuitBreaker(circuitBreaker)
            .get();
    }
}
```

Key design decisions:

Concern	Decision	Rationale
HTTP client	`RestClient` (Spring Boot 4.x)	Modern synchronous client, Micrometer auto-instrumented, virtual-thread-friendly
Timeout	8–10 seconds hard limit	Caller is hearing silence — faster to fail and let Voice Live apologize

Concern	Decision	Rationale
Retry	**No retry**	5 extra seconds of silence is worse than a fast error. Fail fast, let Voice Live handle gracefully
Circuit breaker	Resilience4j	Prevents piling up blocked virtual threads when APIM/backend is down
Authentication	`DefaultAzureCredential` → Bearer token	Azure SDK handles token acquisition, caching, and refresh
Tool registry	`Map<String, ToolDefinition>` from config	Each tool carries APIM endpoint path, timeout override, expected response schema

On failure: `ToolCallException` is thrown, caught by the bridge, and translated to an error tool result for Voice Live:

```
{
  "type": "conversation.item.create",
  "item": {
    "type": "function_call_output",
    "call_id": "call_abc123",
    "output": "{\"error\": true, \"message\": \"Unable to retrieve account balance at this time.\"}"
  }
}
```

Voice Live then speaks a natural error message to the caller.

Accepted: RestClient + Resilience4j circuit breaker, no retry, 8–10s timeout, fail-fast for voice UX.

3.5 Decision 6 — Error Response Model

Recommendation: Sealed exception hierarchy with surface-specific error translation.

Two error surfaces require different handling:

Surface 1 — HTTP responses (webhook controller):

- Event Grid ignores response bodies — status codes only (`401`, `400`, `500`)
- ACS callbacks — RFC 7807 Problem Details (Spring Boot 4.x `ProblemDetail` class) for debugging in logs

Surface 2 — Voice Live tool results (WebSocket):

- Tool call failures → error tool result JSON so Voice Live can inform the caller naturally

Exception hierarchy:

```
public sealed class IvrException extends RuntimeException permits
    UnauthorizedException,    // JWT validation failure → 401
    BadRequestException,      // Malformed event → 400
    ToolCallException,        // Backend call failure → error tool result
    VoiceLiveException {      // Voice Live connection failure → terminate call
}

public final class ToolCallException extends IvrException {
    private final String callId;       // Voice Live tool call ID
    private final String toolName;     // which tool failed
    private final String userMessage;  // safe to speak to caller
}
```

Translation layer:

Exception	HTTP Surface	WebSocket Surface
`UnauthorizedException`	401 Unauthorized	N/A (pre-connection)
`BadRequestException`	400 Bad Request	N/A
`ToolCallException`	N/A	Error tool result JSON
`VoiceLiveException`	N/A	Terminate call via ACS SDK

`@RestControllerAdvice` maps exceptions to HTTP responses. Bridge service maps `ToolCallException` to Voice Live error tool results.

Accepted: Sealed exception hierarchy, `@RestControllerAdvice` for HTTP, bridge-level translation for WebSocket.

3.6 Decision 7 — Dockerfile Strategy

Recommendation: Multi-stage build with Spring Boot layered JAR extraction.

```
# Stage 1: Build
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY .mvn/ .mvn/
COPY mvnw pom.xml ./
RUN ./mvnw dependency:go-offline -B
COPY src/ src/
RUN ./mvnw package -DskipTests -B
RUN java -Djarmode=tools -jar target/*.jar extract --layers --destination
extracted

# Stage 2: Runtime
FROM eclipse-temurin:21-jre
RUN useradd -r appuser
WORKDIR /app
COPY --from=build /app/extracted/dependencies/ ./
```

```
COPY --from=build /app/extracted/spring-boot-loader/ ./
COPY --from=build /app/extracted/snapshot-dependencies/ ./
COPY --from=build /app/extracted/application/ ./
RUN chown -R appuser /app
USER appuser
EXPOSE 8080
HEALTHCHECK CMD curl -f http://localhost:8080/actuator/health/readiness || exit 1
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75", "-jar", "app.jar"]
```

Key properties:

- **Base image:** `eclipse-temurin:21` — Adoptium, regularly patched, no licensing concerns
- **Layered JAR:** Dependencies cached in separate Docker layer (change rarely → fast rebuilds)
- **Non-root user:** `appuser` for container security
- **Health check:** Actuator readiness probe
- **JVM tuning:** `-XX:MaxRAMPercentage=75` for 512MB–1GB container sizing. Container-aware by default in Java 21.
- **Self-contained:** Any CI can run `docker build` — no pre-built JAR required

Options rejected:

- Runtime-only Dockerfile (requires CI to pre-build JAR — less self-contained)
- Standard multi-stage without layers (no Docker layer caching benefit for dependencies)

Accepted: Multi-stage + layered JAR, eclipse-temurin:21, non-root, health check.

3.7 Decision 8 — Kubernetes Manifests

Deferred. Not in scope for this architecture iteration. Will be addressed when deployment pipeline is defined.

3.8 Decision 9 — Configuration Externalization

Recommendation: Environment variables (12-factor) with Spring Boot relaxed binding.

Strategy:

- `application.yaml` for **structural defaults and non-sensitive values** (server port, thread pool sizes, actuator config, logging format)
- Environment variables for **everything environment-specific or sensitive** (tenant ID, ACS connection string, APIM base URL, Voice Live endpoint, tool definitions)
- Secrets flow through: **Azure Key Vault → CSI Secret Store Driver → K8s Secret → env var**

Spring Boot relaxed binding maps env vars to properties automatically:

Environment Variable	Spring Property
`SOFIA_AUTH_TENANT_ID`	`sofia.auth.tenant-id`
`SOFIA_ACS_CONNECTION_STRING`	`sofia.acs.connection-string`
`SOFIA_VOICE_LIVE_ENDPOINT`	`sofia.voice-live.endpoint`

Environment Variable	Spring Property
`SOFIA_APIM_BASE_URL`	`sofia.apim.base-url`
`SOFIA_AUTH_EVENT_GRID_AUDIENCE`	`sofia.auth.event-grid-audience`
`SOFIA_AUTH_ACS_CALLBACK_AUDIENCE`	`sofia.auth.acs-callback-audience`

Options rejected:

- Profile-specific YAML files baked into JAR — secrets in source control, re-deploy to change config
- Azure App Configuration — additional Azure dependency, Spring Cloud Azure dependency, additional startup latency. Worth revisiting if dynamic config refresh becomes a requirement.

Accepted: Environment variables, relaxed binding, Key Vault → CSI → K8s Secret for secrets.

3.9 Decision 10 — Logging Format

Recommendation: JSON structured logging, profile-switched.

Format per environment:

Profile	Format	Reason
`local` / `dev`	Plain text	Human-readable in IDE console
`staging` / `prod`	JSON	Machine-parseable for Azure Monitor / Log Analytics

JSON output example:

```
{
  "timestamp": "2026-02-15T22:45:03.123Z",
  "level": "INFO",
  "logger": "c.s.i.webhook.EventGridController",
  "message": "Incoming call from +5511999887766",
  "traceId": "abc123def456",
  "spanId": "789ghi",
  "callId": "call-xyz-789",
  "sessionId": "sess-abc-123",
  "thread": "virtual-42"
}
```

Custom MDC fields injected per call for correlation:

- `callId` — ACS call ID (filter all logs for one call)
- `sessionId` — Voice Live session ID
- `traceId` / `spanId` — OTel correlation (auto-injected by Micrometer)

Implementation: Logback with `logstash-logback-encoder` for JSON output, profile-switched in `logback-spring.xml`.

Accepted: JSON structured logging, profile-switched, custom MDC fields for call correlation.

Section 4: Implementation Patterns & Consistency Rules

Established in Step 5 — patterns that prevent inconsistency when multiple developers or AI coding agents work on the codebase concurrently.

4.1 Naming Patterns

4.1.1 Java Package Names

Convention: `com.sofia.ivr.<feature>`

Examples: `com.sofia.ivr.webhook`, `com.sofia.ivr.bridge`, `com.sofia.ivr.toolcall`, `com.sofia.ivr.config`

Short, descriptive, one level below the base package. No compound names (`ivrbridge`) or deep nesting.

4.1.2 Class Names

Convention: Short form, no redundant suffixes.

Good	Avoid
`EventGridController`	`EventGridWebhookController`
`AudioBridgeService`	`AudioBridgeServiceImpl`
`ToolCallHandler`	`ToolCallHandlerService`
`AuthProperties`	`AuthenticationProperties`

4.1.3 Configuration Properties Prefix

Convention: `sofia.*` (flat, single-service — no `ivr` nesting).

```
sofia:
  auth:
    event-grid-issuer: "https://login.microsoftonline.com/{tenant}/v2.0"
    acs-issuer: "https://acscallautomation.communication.azure.com"
  voice-live:
    endpoint: "wss://{resource}.openai.azure.com"
    deployment-id: "voice-agent-v1"
```

4.1.4 Test Class & Method Names

- **Test classes:** Singular `…Test` (e.g., `EventGridControllerTest`, not `…Tests`)
- **Test methods:** `should…` BDD-style (e.g., `shouldRejectInvalidJwt()`, `shouldForwardAudioToVoiceLive()`)

4.2 Structural Patterns

4.2.1 Package-by-Feature Layout

Decision: Feature-based package organization (not layer-based).

```
com.sofia.ivr/
├── webhook/          # EventGridController, AcsCallbackController
├── websocket/        # AcsAudioEndpoint (JSR 356 server)
├── bridge/           # AudioBridgeService, VoiceLiveClient
├── toolcall/         # ToolCallHandler, RestClient-based dispatch
├── config/           # AuthProperties, VoiceLiveProperties, SofiaApplication
├── model/            # CallSession, VoiceLive event sealed types
└── exception/        # IvrException sealed hierarchy
```

Rationale: Feature packages map 1:1 to the architecture diagram components. All code for a concern lives together. PRD and copilot-instructions mandate this layout.

4.2.2 Configuration Class Organization

Decision: Single `config/` package for all `@ConfigurationProperties` records.

Configuration is cross-cutting — `AuthProperties` serves webhook, websocket, *and* bridge. Scattering configs across feature packages creates circular-dependency risk.

Structure: Flat records per concern, not a single root object:

```
@ConfigurationProperties(prefix = "sofia.auth")
public record AuthProperties(String eventGridIssuer, String acsIssuer, /* ... */)
{}

@ConfigurationProperties(prefix = "sofia.voice-live")
public record VoiceLiveProperties(String endpoint, String deploymentId, /* ... */)
{}
```

Each feature declares only what it needs as a constructor parameter. No god-object config. Easy to test in isolation.

4.2.3 Shared Model Location

Decision: Top-level `model/` package for domain types shared across features.

Types like `CallSession`, sealed Voice Live event interfaces, and enums live here. The name `model/` is Java-idiomatic. Avoid `common/` (becomes a code-smell magnet).

4.2.4 Exception Class Location

Decision: Dedicated `exception/` package for the sealed `IvrException` hierarchy.

The hierarchy is cross-cutting (thrown from webhook, bridge, toolcall). A separate `exception/` package keeps `model/` focused on data types.

4.2.5 Test Directory Mirroring

Decision: Test packages mirror source packages exactly.

`src/test/java/com/sofia/ivr/webhook/EventGridControllerTest.java` matches `src/main/java/com/sofia/ivr/webhook/EventGridController.java`. Required for package-private access in tests.

4.3 Format Patterns

4.3.1 JSON Field Naming

Convention: camelCase in all REST API responses.

Our REST endpoints interact with ACS (camelCase) and Event Grid (camelCase). Jackson default — zero configuration. Voice Live uses snake_case internally, but the bridge translates at the boundary.

4.3.2 Date/Time Format

Convention: ISO-8601 UTC everywhere (`2026-02-15T14:30:00Z`).

Human-readable in logs, parseable by any client, Spring Boot default for `Instant`/`OffsetDateTime`. Epoch millis used only when Voice Live API requires it.

4.3.3 API Error Response Shape

Convention: RFC 9457 Problem Details (`application/problem+json`).

```
{
  "type": "urn:sofia:ivr:unauthorized",
  "title": "JWT Validation Failed",
  "status": 401,
  "detail": "Token issuer not recognized",
  "instance": "/api/callbacks/events"
}
```

Spring Boot 4.x has built-in `ProblemDetail` class. `@RestControllerAdvice` maps each `IvrException` subclass to a `ProblemDetail` with a single handler method.

4.3.4 WebSocket Message Envelope (Service → Voice Live)

Rule: Voice Live messages are **pass-through** — constructed exactly per the OpenAI Realtime Protocol event schema. No wrapper, no transformation, no custom envelope. Sealed record types model exactly what the wire format requires.

4.3.5 Logging Field Names (MDC Keys)

Convention: OpenTelemetry dot-separated naming.

MDC Key	Example Value
`ivr.call.id`	`call-xyz-789`
`ivr.session.id`	`sess-abc-123`
`ivr.voice_live.session_id`	`vl-sess-456`

Follows OTel semantic conventions (`rpc.method`, `http.request.method`). Auto-instrumented spans use the same style. The OTel exporter translates for Prometheus automatically.

4.4 Communication Patterns

4.4.1 Voice Live Event Type → Java Class Mapping

Convention: Strip dots, PascalCase. Each record carries a `TYPE` constant.

Wire Format	Java Record	Constant
`session.created`	`SessionCreated`	`static final String TYPE = "session.created"`
`response.audio.delta`	`ResponseAudioDelta`	`static final String TYPE = "response.audio.delta"`
`session.update`	`SessionUpdate`	`static final String TYPE = "session.update"`
`input_audio_buffer.append`	`InputAudioBufferAppend`	`static final String TYPE = "input_audio_buffer.append"`

Direct 1:1 traceability between wire format and Java type.

4.4.2 Inbound vs Outbound Event Separation

Decision: Two sealed interfaces.

```
public sealed interface InboundEvent permits SessionCreated, ResponseAudioDelta,
ResponseDone, ToolCall, /* ... */ {}
public sealed interface OutboundEvent permits SessionUpdate,
InputAudioBufferAppend, ToolResult, /* ... */ {}
```

Type safety at the bridge level: `VoiceLiveClient.send(OutboundEvent)` cannot accidentally receive an inbound type. Two small hierarchies are clearer than one large one.

4.4.3 ACS SDK Types — No Wrappers

Rule: ACS SDK types (`AudioData`, `OutStreamingData`, `TranscriptionData`) are used *directly*. No custom wrapper DTOs.

The bridge calls `StreamingData.parse(buffer)` and pattern-matches on the result. Only create custom types if additional context (e.g., `callId` alongside audio bytes) is needed.

4.4.4 Log Message Convention

Convention: Short prose + structured MDC fields.

```
log.info("Voice Live session established for call {}", callId);
log.warn("Tool call failed with status {} for function {}", status, functionName);
log.error("WebSocket closed unexpectedly", exception);
```

MDC fields (`ivr.call.id`, `ivr.session.id`) are auto-populated via a filter/interceptor at WebSocket open and request start. Prose makes logs human-scannable during debugging. Structured fields enable machine queries.

4.4.5 Metric Naming Convention

Convention: OTel dot-separated, lowercase.

Metric	Description
`ivr.call.duration`	Total call duration (seconds)
`ivr.call.active`	Gauge of active calls
`ivr.bridge.audio.bytes_sent`	Counter of bytes ACS→VoiceLive
`ivr.bridge.audio.bytes_received`	Counter of bytes VoiceLive→ACS
`ivr.toolcall.duration`	Histogram of tool call latency
`ivr.toolcall.errors`	Counter of failed tool calls

OTel exporter translates to Prometheus format automatically (dots → underscores, unit suffix appended).

4.5 Process Patterns

4.5.1 WebSocket Lifecycle Management

Decision: Eager Voice Live connection.

1. ACS WebSocket `@OnOpen` fires → bridge creates `CallSession` in `ConcurrentHashMap`
2. Bridge immediately connects to Voice Live via `java.net.http.WebSocket`
3. Bridge sends `session.update` with instructions and tool definitions
4. By the time first audio frame arrives, Voice Live session is ready

Teardown rule: When *either* WebSocket closes (ACS or Voice Live), the bridge tears down the other within 5 seconds. `CallSession` transitions to `TERMINATING` → `TERMINATED`. Resources cleaned up via `try-finally` in the bridge orchestration. `CallSession` removed from `ConcurrentHashMap`.

4.5.2 Audio Bridge Threading Model

Decision: Two virtual threads per call.

- **ACS → Voice Live:** The `@OnMessage` callback on the ACS WebSocket runs on a virtual thread (Tomcat + virtual threads). It reads audio frames and forwards to the Voice Live WebSocket client.
- **Voice Live → ACS:** The `WebSocket.Listener` callbacks (`onText`, `onBinary`) handle inbound Voice Live events. Audio deltas are forwarded to the ACS WebSocket. Tool calls are dispatched to `ToolCallHandler`.

No shared mutable state beyond `CallSession` lookup in `ConcurrentHashMap`. Each direction operates independently. Simple, readable, debuggable.

4.5.3 Validation Timing

Rule: Validate at the boundary, fail fast.

Entry Point	Validation	Failure Response
Webhook controller	JWT validated *before* parsing body	HTTP 401
WebSocket `@OnOpen`	JWT validated in handshake	Close code 4401
Voice Live events	Event type checked against sealed hierarchy	Unknown → WARN log, ignored
Tool call responses	HTTP status from backend validated	Non-2xx → error tool result to Voice Live

Never let invalid data propagate deeper than the boundary.

4.5.4 Graceful Shutdown

Decision: Drain active calls, then stop.

1. Pod receives SIGTERM
2. Health endpoint returns unhealthy → ACS routes new calls elsewhere
3. Active calls continue for up to 30 seconds (`spring.lifecycle.timeout-per-shutdown-phase=30s`)
4. After timeout, remaining WebSockets are force-closed with code 1001 (Going Away)

Configuration: `server.shutdown=graceful` in `application.yaml`.

4.5.5 Idempotency Rules

Component	Rule
Event Grid webhooks	Event Grid may retry. `IncomingCall` handler must be idempotent — if call already answered (exists in `ConcurrentHashMap`), return 200 and ignore.
ACS callbacks	May arrive out of order or duplicate. Check `CallSession` state before acting.

Component	Rule
Tool call dispatch	NOT idempotent — each tool call is a unique invocation. No dedup needed.

4.5.6 WebSocket Close Codes

Code	Meaning	Used When
1000	Normal closure	Call ended gracefully
1001	Going away	Server shutting down
1011	Server error	Unexpected bridge failure
4401	Unauthorized	JWT validation failed on handshake
4408	Timeout	Voice Live connection timed out
4502	Bad gateway	Voice Live refused connection

Custom close codes use the 4xxx range (reserved for applications per RFC 6455). Always include a reason phrase.

4.6 Code Quality Rules

4.6.1 Dependency Injection

Rule: Constructor injection only. No `@Autowired` on fields or setters. With a single constructor, Spring auto-detects — no annotation needed. Records used for `@ConfigurationProperties` get this for free.

4.6.2 Method Visibility

Rule: Package-private by default. Only make `public` what other packages need.

Enforced naturally by feature-package structure — most classes are package-private. Only interfaces exposed to other packages are public. Exception: Spring-annotated classes (`@RestController`, `@Service`, `@Component`) must be `public` for classpath scanning, but their helper methods stay package-private.

4.6.3 Null Handling

Context	Rule
Method parameters (boundary)	`Objects.requireNonNull()`
Return types for "might not exist"	`Optional<T>` (e.g., `Optional<CallSession>` from map lookup)
Record fields (Jackson)	Required fields are non-null by default — Jackson throws on missing
General	Never return `null` from our own code — return `Optional` or throw

4.7 Mandatory Rules (Violations = PR Rejection)

#	Rule	Rationale
M-1	No `Mono`, `Flux`, or reactive types anywhere	Stack is imperative + virtual threads. Mixing reactive creates debugging nightmares.
M-2	No hardcoded secrets, URLs, or connection strings	All externalized via env vars / Key Vault.
M-3	No `Thread.sleep()` for delays	Signals design smell. Use proper timeouts on I/O operations.
M-4	No `synchronized` blocks	Virtual threads pin platform threads inside `synchronized`. Use `ReentrantLock` if mutual exclusion is truly needed.
M-5	No custom thread pools for I/O work	Virtual threads eliminate the need. Let Spring's virtual thread integration handle it.
M-6	No WebSocket audio through HTTP middleware/APIM	Direct WSS connections only. Latency kills voice UX.
M-7	All public REST endpoints must validate JWT	No anonymous endpoints except `/health` and Event Grid validation handshake.
M-8	No `catch (Exception e) {}` — silent swallowing	All exceptions must be logged or propagated. Silent swallowing = invisible bugs.

4.8 Anti-Patterns Specific to This Project

Anti-Pattern	Why It's Tempting	What To Do Instead
Wrapping ACS `AudioData` in custom DTOs	"Clean architecture" instinct	Use SDK types directly — already well-designed
Adding Redis/external cache for call state	"What if we scale?"	ConcurrentHashMap — WebSocket is instance-pinned (ADR-10)
Using Spring Security for JWT	"It's the Spring way"	Nimbus direct — 3 issuers, JSR 356 bypass (ADR-11)
Transcoding audio between ACS and Voice Live	"Maybe formats differ"	Both use PCM 24K mono — pass through raw bytes
Retry on tool call failure	"Resilience!"	Fast-fail + error speech. Retry = 2-8s extra silence.
Creating abstract base classes for events	"DRY!"	Sealed interfaces + records. Composition over inheritance.
WebSocket connection pooling for Voice Live	"Efficiency!"	1:1 call-to-connection. Each call has unique session state.

4.9 Verification Checklist (Code Review / Story Completion)

- ☐ No reactive types in imports (`Mono`, `Flux`, `Publisher`)
- ☐ All config values come from `@ConfigurationProperties` records
- ☐ JWT validated on every inbound HTTP/WebSocket path
- ☐ WebSocket teardown has `try-finally` cleanup
- ☐ Log statements include structured MDC fields
- ☐ No `synchronized` blocks (grep for it)
- ☐ Error responses use `ProblemDetail` (RFC 9457)
- ☐ Tests follow `should…` naming convention
- ☐ Constructor injection only (no `@Autowired` on fields)
- ☐ No `null` returns — use `Optional` or throw
- ☐ Package-private by default, `public` only when needed

Section 5: Project Structure & Boundaries

Established in Step 6 — the physical project layout, file-by-file, and the architectural boundaries between components.

5.1 Requirements → Component Mapping

PRD FR Category	Target Package	Key Responsibilities
FR-CALL (Call Handling)	`webhook/`	Event Grid IncomingCall, answer/reject via `CallAutomationClient`
FR-CALL (Callbacks)	`webhook/`	Mid-call event callbacks from ACS (connected, disconnected, etc.)
FR-AUDIO (Audio Streaming)	`websocket/`	JSR 356 `@ServerEndpoint`, receive/send PCM 24K audio from/to ACS
FR-VOICE (Voice Live)	`bridge/`	Outbound WebSocket to Voice Live, session management, event routing
FR-BRIDGE (Audio Bridge)	`bridge/`	Bidirectional audio forwarding, lifecycle orchestration
FR-TOOL (Tool Calls)	`toolcall/`	Dispatch tool calls to backends via APIM, return results to Voice Live
FR-SEC (Security)	Cross-cutting → `config/`	JWT validation (Nimbus), managed identity credential
FR-OBS (Observability)	Cross-cutting → `config/`	OTel autoconfig, MDC filter, health endpoint
FR-STATE (Call State)	`model/`	`CallSession` record, state machine (12 states, 33 transitions)
FR-CONFIG (Configuration)	`config/`	61 config params, `@ConfigurationProperties` records

5.2 Cross-Cutting Concerns → Locations

Concern	Location
JWT validation	`config/JwtValidatorFactory.java` + `config/JwtValidator.java` — used in webhook filters and WebSocket handshake
MDC propagation	`config/MdcFilter.java` (servlet filter) + manual MDC set in WebSocket `@OnOpen`
Error handling	`exception/` hierarchy + `config/GlobalExceptionHandler.java` (`@RestControllerAdvice`)
Azure credential	`config/AzureCredentialConfig.java` — single `DefaultAzureCredential` bean
Health check	`config/HealthIndicator.java` or Spring Boot Actuator auto-config

5.3 Complete Project Directory Structure

```
sofia-ivr-bridge/
├── .github/
│   ├── copilot-instructions.md              # AI coding agent guidelines
│   └── workflows/
│       └── ci.yml                            # GitHub Actions CI pipeline
├── .gitignore
├── .editorconfig                             # Consistent formatting across IDEs
├── README.md
├── CHANGELOG.md
├── Dockerfile                                # Multi-stage, layered JAR (Decision
7)
├── docker-compose.yml                        # Local dev: service + Azurite +
mock backends
├── pom.xml                                   # Maven POM (BOM ordering per Step
3)
├── mvnw / mvnw.cmd / .mvn/                   # Maven wrapper
│
├── src/
│   ├── main/
│   │   ├── java/com/sofia/ivr/
│   │   │   │
│   │   │   ├── SofiaIvrApplication.java                  # @SpringBootApplication
entry point
│   │   │   │
│   │   │   ├── webhook/                                   # --- Call Handling &
Callbacks ---
│   │   │   │   ├── EventGridController.java               # POST /api/events —
IncomingCall + validation handshake + admission control guard (REQ-R4.1)
│   │   │   │   ├── AcsCallbackController.java             # POST
/api/callbacks/events — mid-call events
│   │   │   │   ├── CallAutomationService.java             # Wraps
```

```
CallAutomationClient (answer, hangup, play)
│   │   │   │   └── EventGridEvent.java                    # Minimal DTO for Event
Grid envelope parsing
│   │   │   │
│   │   │   ├── websocket/                                 # --- ACS Audio
WebSocket Server ---
│   │   │   │   ├── AcsAudioEndpoint.java                  # @ServerEndpoint — PCM
24K in/out from ACS
│   │   │   │   ├── AcsAudioEndpointConfig.java            #
ServerEndpointExporter bean + auth handshake
│   │   │   │   └── AcsHandshakeInterceptor.java           # JWT validation + per-
IP rate limiting (SEC-REQ-15) on WebSocket upgrade
│   │   │   │
│   │   │   ├── bridge/                                    # --- Audio Bridge &
Voice Live Client ---
│   │   │   │   ├── AudioBridgeService.java                # Orchestrates
bidirectional audio forwarding
│   │   │   │   ├── VoiceLiveClient.java                   #
java.net.http.WebSocket client to Voice Live
│   │   │   │   ├── VoiceLiveListener.java                 # WebSocket.Listener —
handles inbound VL events
│   │   │   │   └── VoiceLiveSessionInitializer.java       # Sends session.update
with instructions + tools
│   │   │   │
│   │   │   ├── toolcall/                                  # --- Tool Call
Dispatch ---
│   │   │   │   ├── ToolCallHandler.java                   # Routes tool_call
events to backend APIs
│   │   │   │   ├── ToolCallDispatcher.java                # RestClient +
Resilience4j circuit breaker
│   │   │   │   └── ToolDefinitionRegistry.java            # Loads tool
definitions for session.update
│   │   │   │
│   │   │   ├── model/                                     # --- Domain Types ---
│   │   │   │   ├── CallSession.java                       # Call state record
(per CC-2 state machine)
│   │   │   │   ├── CallState.java                         # Enum: INITIALIZING,
STREAMING, TERMINATING, TERMINATED (CC-2)
│   │   │   │   ├── CallSessionStore.java                  # ConcurrentHashMap
wrapper with Optional returns
│   │   │   │   ├── InboundEvent.java                      # sealed interface —
Voice Live → Service events
│   │   │   │   ├── OutboundEvent.java                     # sealed interface —
Service → Voice Live events
│   │   │   │   ├── SessionCreated.java                    # record implements
InboundEvent
│   │   │   │   ├── ResponseAudioDelta.java                # record implements
InboundEvent
│   │   │   │   ├── ResponseDone.java                      # record implements
InboundEvent
│   │   │   │   ├── ToolCallEvent.java                     # record implements
InboundEvent
│   │   │   │   ├── SessionUpdate.java                     # record implements
OutboundEvent
```

```
│   │   │   ├── InputAudioBufferAppend.java          # record implements
OutboundEvent
│   │   │   ├── ToolResult.java                      # record implements
OutboundEvent
│   │   │   └── VoiceLiveEventParser.java            # Jackson
deserialization → sealed type dispatch
│   │   │
│   │   ├── exception/                               # --- Error Hierarchy -
--
│   │   │   ├── IvrException.java                    # sealed abstract class
│   │   │   ├── UnauthorizedException.java           # extends IvrException
(4xx)
│   │   │   ├── BadRequestException.java             # extends IvrException
(4xx)
│   │   │   ├── ToolCallException.java               # extends IvrException
(tool dispatch failures)
│   │   │   └── VoiceLiveException.java              # extends IvrException
(VL connection failures)
│   │   │
│   │   └── config/                                  # --- Configuration &
Cross-Cutting ---
│   │       ├── AuthProperties.java                  #
@ConfigurationProperties(prefix = "sofia.auth")
│   │       ├── VoiceLiveProperties.java             #
@ConfigurationProperties(prefix = "sofia.voice-live")
│   │       ├── AcsProperties.java                   #
@ConfigurationProperties(prefix = "sofia.acs")
│   │       ├── ToolCallProperties.java              #
@ConfigurationProperties(prefix = "sofia.toolcall")
│   │       ├── JwtValidatorFactory.java             # Creates validators
per issuer (Nimbus)
│   │       ├── JwtValidator.java                    # Validates + extracts
claims for a specific issuer
│   │       ├── AzureCredentialConfig.java           #
DefaultAzureCredential bean
│   │       ├── RestClientConfig.java                # RestClient +
Resilience4j bean configuration
│   │       ├── WebSocketConfig.java                 #
ServerEndpointExporter + ObjectMapper config
│   │       ├── GlobalExceptionHandler.java          # @RestControllerAdvice
→ ProblemDetail
│   │       ├── MdcFilter.java                       # Servlet filter:
populates MDC for HTTP requests
│   │       └── ObservabilityConfig.java             # OTel custom metric
beans, Micrometer config
│   │
│   └── resources/
│       ├── application.yaml                         # Default config (all
profiles)
│       ├── application-local.yaml                   # Local dev overrides
│       ├── application-staging.yaml                 # Staging overrides (if
needed)
│       ├── application-prod.yaml                    # Prod overrides (if
needed)
```

```
│   │       ├── logback-spring.xml                          # Profile-switched:
plain text (local) / JSON (prod)
│   │       └── tool-definitions/                           # Tool JSON schemas for
Voice Live session.update
│   │           ├── get-user-data.json                      # GET user/account data
(PRD tool #1)
│   │           ├── send-invoice-registered-email.json      # Send invoice to
registered email (PRD tool #2)
│   │           └── send-invoice-provided-email.json        # Send invoice to user-
provided email (PRD tool #3)
│   │
│   └── test/
│       ├── java/com/sofia/ivr/
│       │   │
│       │   ├── webhook/
│       │   │   ├── EventGridControllerTest.java           # JWT validation,
subscription handshake, IncomingCall
│       │   │   ├── AcsCallbackControllerTest.java         # Callback event
handling, idempotency
│       │   │   └── CallAutomationServiceTest.java         # Mock ACS SDK
interactions
│       │   │
│       │   ├── websocket/
│       │   │   ├── AcsAudioEndpointTest.java              # WebSocket lifecycle,
audio frame handling
│       │   │   └── AcsHandshakeInterceptorTest.java       # JWT rejection on
upgrade
│       │   │
│       │   ├── bridge/
│       │   │   ├── AudioBridgeServiceTest.java            # Bidirectional
forwarding, teardown
│       │   │   ├── VoiceLiveClientTest.java               # Connection, send,
reconnect scenarios
│       │   │   └── VoiceLiveListenerTest.java             # Event parsing and
dispatch
│       │   │
│       │   ├── toolcall/
│       │   │   ├── ToolCallHandlerTest.java               # Routing logic, error
handling
│       │   │   └── ToolCallDispatcherTest.java            # Circuit breaker,
timeout, error responses
│       │   │
│       │   ├── model/
│       │   │   ├── CallSessionStoreTest.java              # Concurrent access,
idempotent put
│       │   │   ├── CallStateTest.java                     # State transition
validation
│       │   │   └── VoiceLiveEventParserTest.java          # Deserialization of
all event types
│       │   │
│       │   ├── exception/
│       │   │   └── GlobalExceptionHandlerTest.java        # ProblemDetail mapping
for each exception type
│       │   │
```

```
│       │   ├── config/
│       │   │   ├── JwtValidatorFactoryTest.java          # Multi-issuer
validator creation
│       │   │   └── JwtValidatorTest.java                 # Token validation per
issuer type
│       │   │
│       │   └── integration/                              # Integration tests
(Spring Boot @SpringBootTest)
│       │       ├── WebhookIntegrationTest.java           # Full HTTP roundtrip
with embedded server
│       │       ├── WebSocketIntegrationTest.java         # WebSocket connect +
audio frame exchange
│       │       └── ToolCallIntegrationTest.java          # RestClient → WireMock
backend
│       │
│       └── resources/
│           ├── application-test.yaml                     # Test profile config
│           ├── test-jwks.json                            # Test JWKS for JWT
validation tests
│           ├── fixtures/                                 # Test data
│           │   ├── event-grid-incoming-call.json
│           │   ├── acs-callback-connected.json
│           │   ├── voice-live-session-created.json
│           │   ├── voice-live-tool-call.json
│           │   └── audio-frame-pcm24k.bin
│           └── wiremock/                                 # WireMock stubs for
integration tests
│               ├── check-balance-200.json
│               └── get-customer-info-500.json
│
├── docs/                                                 # Project documentation
│   └── (architecture diagrams, runbooks, etc.)
│
└── planning/                                             # BMAD planning
artifacts
    ├── planning-artifacts/
    │   ├── prd.md
    │   └── architecture.md
    └── implementation-artifacts/
        └── (stories, sprint plans — generated later)
```

5.4 Architectural Boundaries

5.4.1 Inbound API Boundaries

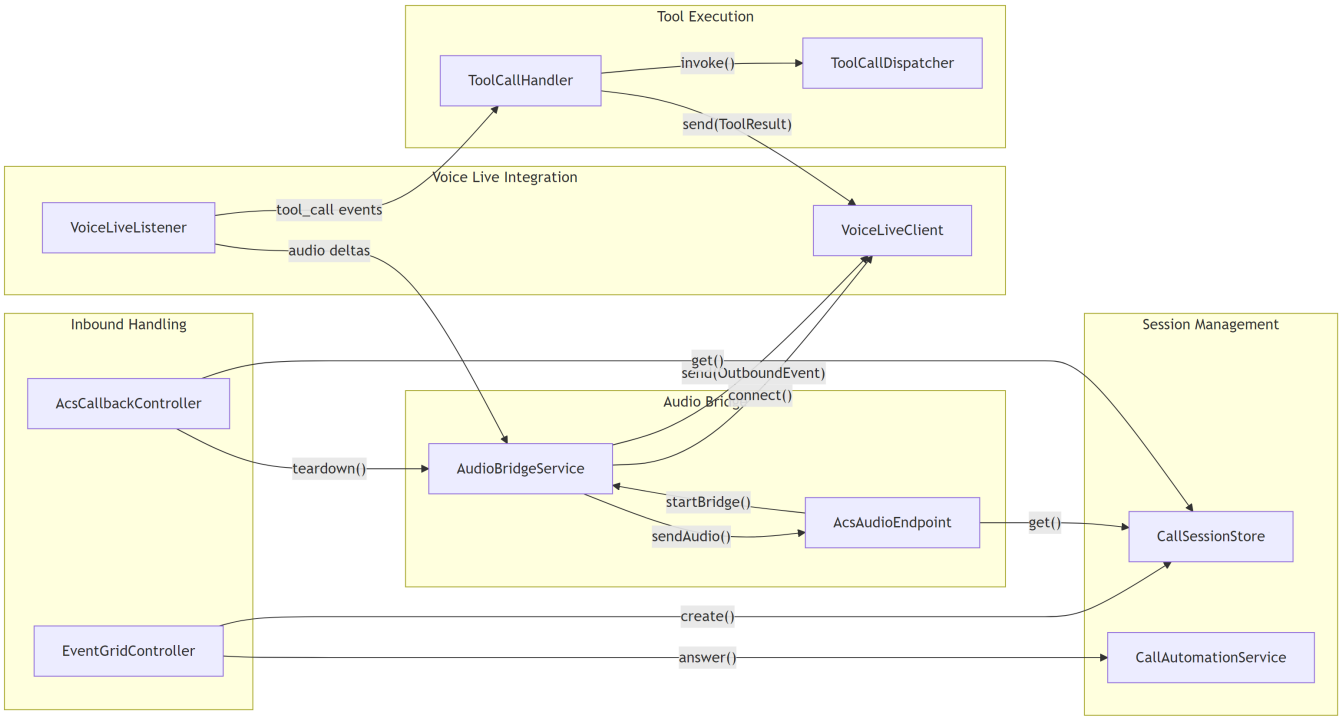
Endpoint	Controller	Auth	Purpose
`POST /api/events`	`EventGridController`	Entra ID JWT	Event Grid subscriptions + IncomingCall
`POST /api/callbacks/events`	`AcsCallbackController`	ACS signed JWT	Mid-call events (connected, disconnected, etc.)

Endpoint	Controller	Auth	Purpose
`WSS /ws/audio`	`AcsAudioEndpoint`	ACS JWT in handshake	Bidirectional PCM 24K audio
`GET /actuator/health`	Spring Boot Actuator	None (anonymous)	Liveness/readiness probes

5.4.2 Outbound Integration Boundaries

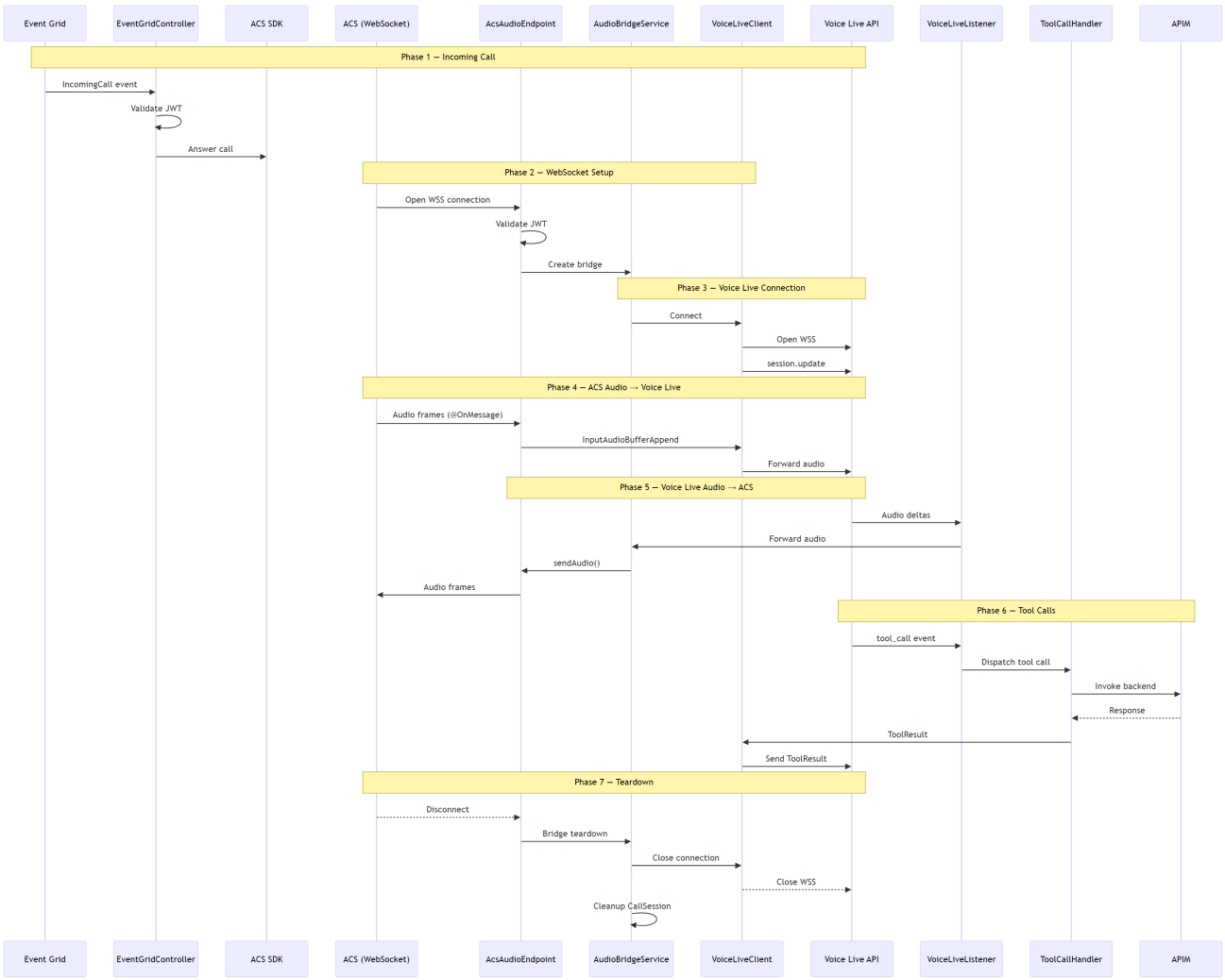
Target	Client	Auth	Purpose
Voice Live API	`VoiceLiveClient` (JDK WebSocket)	`DefaultAzureCredential`	Real-time voice agent
APIM → Backend APIs	`ToolCallDispatcher` (RestClient)	`DefaultAzureCredential`	Tool call dispatch
ACS Call Automation	`CallAutomationService` (ACS SDK)	Connection string / managed identity	Answer, hangup, play

5.4.3 Internal Component Communication



Source: component-communication.mmd

5.4.4 Data Flow — Single Call Lifecycle



Source: call-lifecycle.mmd

5.5 File Organization Patterns

5.5.1 Configuration Files

File	Purpose
`application.yaml`	Shared defaults across all profiles
`application-local.yaml`	Local dev (localhost URLs, debug logging)
`application-staging.yaml`	Staging environment overrides
`application-prod.yaml`	Production (minimal — most config from env vars / Key Vault)
`logback-spring.xml`	Logging format switching (plain text vs JSON by profile)

5.5.2 Test Organization

Directory	Purpose	Framework
`test/.../webhook/`	Unit tests for controllers and services	JUnit 5 + Mockito
`test/.../websocket/`	Unit tests for WebSocket endpoint	JUnit 5 + Mockito

Directory	Purpose	Framework
`test/.../bridge/`	Unit tests for bridge and VL client	JUnit 5 + Mockito
`test/.../toolcall/`	Unit tests for tool dispatch	JUnit 5 + Mockito
`test/.../model/`	Unit tests for domain types and parsing	JUnit 5
`test/.../config/`	Unit tests for JWT validators	JUnit 5 + Nimbus test keys
`test/.../integration/`	Integration tests with Spring context	`@SpringBootTest` + WireMock
`test/resources/fixtures/`	JSON fixtures for event parsing tests	Static test data
`test/resources/wiremock/`	WireMock stubs for backend API mocking	WireMock JSON mapping

5.5.3 Tool Definitions

Tool JSON schemas live in `src/main/resources/tool-definitions/`. Each file defines one tool per the OpenAI function calling schema. `ToolDefinitionRegistry` loads them at startup and includes them in `session.update` sent to Voice Live.

Section 6: Architecture Validation Results

Established in Step 7 — comprehensive validation of coherence, requirements coverage, implementation readiness, and gap resolution.

6.1 Coherence Validation ☑

Decision Compatibility: PASS

All 10 architectural decisions are internally consistent:

- Java 21 + Spring Boot 4.0.x + Tomcat + virtual threads form a coherent imperative stack
- JSR 356 `@ServerEndpoint` runs on Tomcat's WebSocket subsystem — compatible with virtual thread executor
- `java.net.http.WebSocket` is JDK-native, no library conflicts
- Nimbus JOSE+JWT is a Spring Boot transitive dependency — zero new JARs added for JWT (ADR-11)
- OTel Spring Boot Starter autoconfig (ADR-9) works with Spring Boot 4.x and Micrometer
- `DefaultAzureCredential` via `azure-identity` is framework-agnostic — works in all 3 auth contexts
- ConcurrentHashMap (ADR-10) aligns with virtual thread model — no pinning risk, lock-free reads
- Resilience4j integrates with Spring Boot 4.x autoconfiguration and RestClient
- ADRs 7–11 reinforce each other with no contradictions

Pattern Consistency: PASS

- Naming conventions (§4.1) consistent across all file tree entries and examples
- Configuration prefix `sofia.*` used uniformly in all `@ConfigurationProperties` annotations and `application.yaml` examples

- Note: The PRD Configuration Catalog uses `ivr.*` prefix (C-01 through C-60) as requirements-level identifiers. The architecture standardizes implementation on `sofia.*`. Mapping: PRD `ivr.acs.*` → code `sofia.acs.*`, PRD `ivr.voicelive.*` → code `sofia.voice-live.*`, etc. AI agents implementing stories should translate PRD C-xx references to `sofia.*` properties.
- JSON format (camelCase), date format (ISO-8601 UTC), error shape (RFC 9457), and logging MDC keys (OTel dot-separated) are consistent throughout
- Voice Live wire format pass-through rule (§4.3.4) aligns with sealed record modeling (§4.4.1)

Structure Alignment: PASS

- Project structure (~60 files, 7 feature packages) maps directly to architectural component diagram
- Boundaries (4 inbound endpoints, 3 outbound integrations) match §5.4 exactly
- Test directory mirrors source packages for package-private access
- Integration test directory covers the 3 critical boundaries (webhook, WebSocket, tool call)

6.2 Requirements Coverage Validation ☑

Functional Requirements (50 FRs across 10 Capabilities): COVERED

FR Category	Architectural Coverage
CAP-1 Call Handling (6 FRs)	`webhook/EventGridController` + `CallAutomationService` — answer, reject, callback handling
CAP-2 Audio Streaming (6 FRs)	`websocket/AcsAudioEndpoint` + `bridge/AudioBridgeService` — PCM 24K bidirectional forwarding
CAP-3 Voice AI Integration (6 FRs)	`bridge/VoiceLiveClient` + `VoiceLiveListener` + `VoiceLiveSessionInitializer`
CAP-4 Session Management (5 FRs)	`model/CallSession` + `CallSessionStore` + CC-2 state machine
CAP-5 Tool Call Framework (5 FRs)	`toolcall/ToolCallHandler` + `ToolCallDispatcher` + `ToolDefinitionRegistry`
CAP-6 Security (5 FRs)	`config/JwtValidatorFactory` + `JwtValidator` + ADR-11 reference implementation
CAP-7 Error Handling (5 FRs)	`exception/` sealed hierarchy + `GlobalExceptionHandler` + CC-4 degradation
CAP-8 Observability (5 FRs)	`config/ObservabilityConfig` + OTel autoconfig (ADR-9) + §4.4.5 metric naming
CAP-9 Configuration (5 FRs)	`config/*Properties` records + §4.1.3 prefix convention + §4.2.2 flat records
CAP-10 Call Transfer (2 FRs)	Intentionally deferred to Growth phase — acknowledged, not a gap

Non-Functional Requirements (47 NFRs): COVERED

- **Performance (10):** Virtual threads eliminate thread bottleneck (ADR-8); audio path purity (TC-17) ensures zero-overhead forwarding; eager VL connection (§4.5.1) reduces time-to-first-audio
- **Reliability (8):** CC-4 degradation hierarchy; CC-6 deterministic resource cleanup; CC-7 correlated failure isolation; §4.5.4 graceful shutdown
- **Security (10):** ADR-11 covers all 3 inbound auth paths with reference implementation; `DefaultAzureCredential` everywhere (TC-6); zero audio storage (TC-12); RFC 9457 error hardening (§4.3.3); admission control in `EventGridController`; rate limiting in `AcsHandshakeInterceptor`
- **Scalability (6):** ConcurrentHashMap lock-free reads (ADR-10); virtual threads scale to 10K+ (ADR-8); container-portable (AP-4)
- **Observability (8):** OTel autoconfig Day 1 (ADR-9); structured JSON logging via logstash-logback-encoder; 16 Micrometer metrics; 23 log events; 5 OTel spans; CC-1 correlation propagation
- **Maintainability (5):** Config-driven prompts (AP-6); feature packages (§4.2.1); sealed types for exhaustive pattern matching

Security Requirements (18 SEC-REQs): COVERED

Priority	SEC-REQs	Status
P0 (10)	SEC-REQ-1, 2, 3, 4, 5, 9, 10, 11, 14, 15	All have explicit architectural support
P1 (8)	SEC-REQ-6, 7, 8, 12, 13, 16, 17, 18	Acknowledged, intentionally deferred

UX Requirements (11 UX-REQs): COVERED

All map to process patterns (§4.5) or cross-cutting concerns (CC-2, CC-4). Fallback prompts externalized as config (C-35 through C-39).

Acceptance Criteria (24 ACs): COVERED

All 24 acceptance criteria are testable with the defined test infrastructure (FakeAcsClient + FakeVoiceLiveServer + WireMock + integration tests).

6.3 Implementation Readiness Validation ☑

Decision Completeness: HIGH

- All 10 decisions documented with versions, rationale, alternatives rejected, and "Accepted" statements
- ADR-11 includes a **full reference implementation** (~100 lines) for the hardest auth problem (3 JWT validation paths)
- ADR-10 includes capacity analysis with latency comparison (50ns vs 500µs–2ms)
- All dependencies listed with sources and artifact names (including Resilience4j and logstash-logback-encoder)

Structure Completeness: HIGH

- ~60 files with per-file purpose comments in the directory tree
- §5.1 maps FR categories to packages; §5.2 maps cross-cutting concerns to locations
- §5.4 defines all 4 inbound and 3 outbound boundaries
- §5.5 specifies config files, test organization, and tool definitions

Pattern Completeness: HIGH

- 9 pattern subsections covering naming, structure, format, communication, process, code quality
- 8 mandatory rules (M-1 to M-8) with violation = PR rejection
- 7 anti-patterns specific to this project
- 11-item verification checklist
- Admission control: `EventGridController` checks `CallSessionStore.size()` against `sofia.resilience.admission.max-calls` before calling `answerCall()`. Above threshold → answer briefly, play busy prompt, hangup (UX-REQ-11)
- WebSocket rate limiting: `AcsHandshakeInterceptor` maintains a `ConcurrentHashMap<String, AtomicInteger>` of per-IP connection counts with TTL sweep. Exceeds `sofia.security.websocket.rate-limit-per-ip` → reject upgrade with HTTP 429

6.4 Gap Analysis Results

Six gaps were identified during validation. All were **alignment details**, not structural deficiencies. Resolutions applied:

#	Gap	Resolution
G-1	Config prefix `sofia.*` (architecture) vs `ivr.*` (PRD)	**Kept `sofia.*`** as the implementation prefix. Documented the PRD→code mapping in §6.1 Pattern Consistency. AI agents translate PRD C-xx references to `sofia.*` properties.
G-2	Admission control had no architectural component	Added to `EventGridController` file-tree annotation and documented pattern in §6.3
G-3	WebSocket per-IP rate limiting had no pattern	Added to `AcsHandshakeInterceptor` file-tree annotation and documented pattern in §6.3
G-4	`CallState.java` file-tree comment listed wrong enum values	Updated to match CC-2: INITIALIZING, STREAMING, TERMINATING, TERMINATED
G-5	Resilience4j and logstash-logback-encoder missing from §2.4	Added to dependency table
G-6	Tool definition filenames didn't match PRD's 3 tools	Updated to `get-user-data.json`, `send-invoice-registered-email.json`, `send-invoice-provided-email.json`

6.5 Architecture Completeness Checklist

☑ Requirements Analysis (Section 1)

- ☑ Project context thoroughly analyzed — 50 FRs, 47 NFRs, 61 config params
- ☑ Scale and complexity assessed — 3 architectural tiers, PCM 24K audio path
- ☑ Technical constraints identified — 17 constraints (TC-1 to TC-17)
- ☑ Cross-cutting concerns mapped — 7 concerns (CC-1 to CC-7)

☑ Starter Template (Section 2)

- ☑ Technology domain identified — Java backend service
- ☑ Starter selected with rationale — Initializr + ACS Sample reference
- ☑ All dependencies listed with sources and BOM ordering
- ☑ Initialization command provided

☑ Architectural Decisions (Section 3)

- ☑ 10 critical decisions documented with versions
- ☑ 5 ADRs recorded (ADR-7 through ADR-11)
- ☑ Technology stack fully specified
- ☑ Integration patterns defined (WebSocket server + client, RestClient + APIM)
- ☑ Performance considerations addressed (audio path purity, eager connection, virtual threads)
- ☑ Reference implementation for highest-risk decision (ADR-11 JWT validation)

☑ Implementation Patterns (Section 4)

- ☑ Naming conventions established (§4.1 — 4 subsections)
- ☑ Structure patterns defined (§4.2 — 5 subsections)
- ☑ Format patterns specified (§4.3 — 5 subsections)
- ☑ Communication patterns documented (§4.4 — 5 subsections)
- ☑ Process patterns complete (§4.5 — 6 subsections)
- ☑ Code quality rules defined (§4.6)
- ☑ Mandatory rules enumerated (§4.7 — M-1 to M-8)
- ☑ Anti-patterns catalogued (§4.8 — 7 items)
- ☑ Verification checklist ready (§4.9 — 11 items)

☑ Project Structure (Section 5)

- ☑ Complete directory structure defined (~60 files)
- ☑ Component boundaries established (4 inbound, 3 outbound)
- ☑ Integration points mapped with data flow diagram
- ☑ Requirements-to-structure mapping complete (§5.1)
- ☑ Cross-cutting concern locations specified (§5.2)

6.6 Architecture Readiness Assessment

Overall Status: READY FOR IMPLEMENTATION

Confidence Level: HIGH — all decisions validated, all requirements covered, all gaps resolved

Key Strengths:

1. **Audio path purity** — TC-17, CC-2, and tiering ensure the real-time audio path has zero business logic contamination
2. **Reference implementation for hardest problem** — ADR-11 provides ~100 lines of production-ready JWT validation code covering all 3 inbound auth paths
3. **Sealed type system** — InboundEvent/OutboundEvent sealed interfaces + IvrException sealed hierarchy give AI agents exhaustive pattern matching with compiler enforcement

4. **Virtual thread alignment** — every decision (ConcurrentHashMap over Redis, ReentrantLock over synchronized, RestClient blocking calls) is designed for Project Loom
5. **Mandatory rules as guardrails** — M-1 to M-8 prevent the most common mistakes before they enter the codebase
6. **Comprehensive test infrastructure** — FakeAcsClient + FakeVoiceLiveServer + WireMock enable cloud-free testing from Day 1

Areas for Future Enhancement:

- K8s manifests (intentionally deferred — Decision 8)
- Call transfer architecture (CAP-10, Growth phase)
- PII masking implementation (SEC-REQ-6, P1)
- KEDA autoscaling rules
- Session resumption semantics for Voice Live reconnection

6.7 PRD Configuration Prefix Mapping

The PRD Configuration Catalog uses `ivr.*` prefix (C-01 through C-60). The architecture standardizes on `sofia.*` for implementation. Translation table for AI implementing agents:

PRD Prefix	Code Prefix	`@ConfigurationProperties`
`ivr.acs.*`	`sofia.acs.*`	`AcsProperties`
`ivr.voicelive.*`	`sofia.voice-live.*`	`VoiceLiveProperties`
`ivr.security.*` / `ivr.security.jwt.*`	`sofia.auth.*`	`AuthProperties`
`ivr.resilience.*`	`sofia.resilience.*`	`ResilienceProperties`
`ivr.call.*`	`sofia.call.*`	`CallProperties`
`ivr.audio.*`	`sofia.audio.*`	`AudioProperties`
`ivr.shutdown.*`	`sofia.shutdown.*`	`ShutdownProperties`
`ivr.prompts.*`	`sofia.prompts.*`	`PromptsProperties`
`ivr.tool-call.*`	`sofia.toolcall.*`	`ToolCallProperties`
`ivr.websocket.*`	`sofia.websocket.*`	`WebSocketProperties`
`management.*`	`management.*`	Spring Boot native (unchanged)

Metric names remain `ivr.*` (e.g., `ivr.call.duration`) — these are observability identifiers, not Spring config properties.

6.8 Implementation Handoff

AI Agent Guidelines:

- Follow all architectural decisions exactly as documented in Sections 1–5
- Use implementation patterns (Section 4) consistently across all components
- Respect project structure and boundaries (Section 5)

- Translate PRD `ivr.*` config references to `sofia.*` per §6.7
- Refer to this document for all architectural questions
- ADR-11 reference implementation is production-ready — use it as the starting point for JWT validation

First Implementation Step:

```
curl https://start.spring.io/starter.zip \
  -d type=maven-project \
  -d language=java \
  -d javaVersion=21 \
  -d bootVersion=4.0.0 \
  -d groupId=com.sofia \
  -d artifactId=ivr-bridge \
  -d name=sofia-ivr-bridge \
  -d description="ACS Voice Live IVR Bridge Service" \
  -d packageName=com.sofia.ivr \
  -d dependencies=web,websocket,actuator,validation,configuration-processor \
  -o ivr-bridge.zip
```

Then apply the package layout from §5.3 and configure `spring.threads.virtual.enabled=true`.