Flingoos Desktop Service - Implementation Roadmap

Session-based desktop UI for triggering data collection, processing, and workflow display

> **Integrates with existing Flingoos Bridge installer, auto-updater, and distribution system Follows established phase-based development methodology Leverages existing JWT + GCP + Firebase Auth security architecture**

System Architecture & Data Flow

```
graph TD
    subgraph "Multi-Tenant Desktop Service Architecture"

        UI[Desktop UI]
        DS[Desktop Service - on-premise]

        subgraph "Session Recording - Existing Bridge"
            BA[Bridge API - on-premise]
            Bridge[Flingoos Bridge Session Manager]
            GCP[(GCP Storage - cloud)]
        end

        subgraph "Processing Pipeline - Local Forge CLI"
            FA[Backend Worker - cloud run]
            Forge[Forge CLI Tool - local execution]
            FS[(Firestore - cloud)]
        end

        subgraph "Authentication - Existing"
            Auth[JWT + Firebase Auth + GCP Signed URLs]
        end
    end

    %% Data Flow with Step Labels
    UI -->|"A. Start Recording"| DS
    DS -->|"B. POST /session/start"| BA
    BA -->|"C. Enable collectors"| Bridge
    Bridge -->|"D. Collect & upload data"| GCP

    UI -->|"E. Stop Recording"| DS
    DS -->|"F.1 POST /session/stop"| BA
        DS -->|"F.2 POST /sessions/{id}/process"| FA

    BA -->|"G. Disable collectors"| Bridge
    FA -->|"H. Execute forge CLI"| Forge
    Forge -->|"I. Download session data"| GCP
    Forge -->|"J. Process & upload workflow"| FS

    DS -->|"K. Poll workflow status"| FA
    FA -->|"L. Check Firestore"| FS
    DS -->|"M. Fetch completed workflow"| FS
    DS -->|"N. Display results"| UI

    %% Security connections
    DS -.->|"JWT Tokens"| Auth
    BA -.->|"JWT Validation"| Auth
    FA -.->|"JWT Validation"| Auth
```

Implementation Strategy

Development Diary Methodology: Following established phase-based approach with numbered entries, standard format (Decision/Challenge, Context, Resolution, Impact, Follow-up)

Distribution Integration: Desktop service integrated into existing Inno Setup installer, PyInstaller build system, and auto-updater architecture

Security Continuity: Leverages existing JWT + GCP Signed URLs + Firebase Auth without modification

---

⚠️ Critical Integration: Desktop Service Distribution

ISSUE IDENTIFIED: Desktop service must integrate with existing sophisticated installer/updater system rather than being a separate application.

Current Flingoos Bridge Distribution System:

- **Inno Setup Installer** (`installer.iss`): Windows installer with admin privileges, user configuration, auto-start
- **PyInstaller Build** (`flingoos-bridge.spec`): Windows executable with all dependencies bundled
- **Go Auto-Updater** (`updater/main.go`): Secure auto-update with signature verification, UAC elevation
- **Build Automation** (`scripts/build-installer.ps1`): Complete build pipeline with testing, signing, distribution
- **Code Signing**: Self-signed certificates for development, production signing pipeline

Desktop Service Integration Strategy:

1. **Desktop Service as Bridge Component**: Integrate desktop service into existing bridge codebase as `bridge.ui.desktop` module
2. **Shared Installer**: Extend `installer.iss` to include desktop service UI components and dependencies
3. **Unified Auto-Update**: Desktop service updates with bridge via existing updater system
4. **Consistent Versioning**: Desktop service version matches bridge version in `config.toml`
5. **Single Installation**: Users install one application with both bridge and desktop service capabilities

Updated Installation Flow:

```
User runs flingoos-bridge-installer.exe
    ↓
Installs: bridge.exe + desktop-service (web UI) + updater.exe
    ↓
User launches bridge.exe --desktop-mode
    ↓
Bridge starts with desktop UI server + existing bridge functionality
```

This resolves: Version management, distribution complexity, user confusion, and maintenance overhead.

Implementation Overview

Following established development diary methodology with numbered entries and standard format

Phase	Repository	Focus Area	Key Deliverables
Phase 1	`flingoos-bridge`	Session API Integration	REST API, Session Management, Database Schema
Phase 2	`flingoos-bridge-backend-service`	Forge Processing Integration	Session Processing, Job Tracking, Status API
Phase 3	`flingoos-bridge`	Desktop UI Integration	Web UI, Service Orchestration, Authentication
Phase 4	`flingoos-bridge`	Installer & Distribution	PyInstaller, Inno Setup, Auto-Updater Integration
Phase 5	All repositories	Testing & Deployment	E2E Testing, Security Validation, Production Deploy

Multi-tenant, enterprise-grade security | Seamless integration with existing systems

---

⚠️ Critical Architecture Update: Forge CLI Integration

Issue Discovered: Initial analysis incorrectly described Forge as a web service with REST endpoints.

Corrected Understanding: **Flingoos-Forge is a local CLI tool** that: - Downloads raw data from GCS buckets (where Bridge uploads session data) - Processes data locally through stages A-F (segmentation, annotation, transcription, LLM analysis) - Uploads processed workflows directly to Firestore using service account credentials

Updated Integration Approach: 1. Backend service receives `POST /sessions/{id}/process` request 2. Backend spawns local `forge analyze` command with session parameters
3. Forge CLI downloads data from GCS, processes locally, uploads to Firestore 4. Desktop service polls backend for completion status 5. Desktop service fetches completed workflow from Firestore

Authentication: Forge uses GCS service account credentials (not JWT tokens) and Firestore service account for uploads.

⚠️ Critical Architecture Update: Simplified Credentialization Model

Issue Discovered: Initial analysis incorrectly described Forge as a web service and overcomplicated device authorization.

Corrected Understanding: 1. **Flingoos-Forge is a local CLI tool** (not a web service) 2. **Device Authorization**: Current device only - user analyzes their own device data 3. **Device Identity**: Use exact same mechanism as Bridge for consistency

Updated Integration Approach: 1. Desktop UI auto-detects device_id using Bridge's `DeviceIdentity.get_device_id()` 2. Backend receives session request with user JWT (device_id implicit from request context) 3. Backend spawns `forge analyze` command with user's device only 4. Forge processes user's device data and uploads to Firestore with user context 5. Desktop service retrieves user's workflows from Firestore

Authentication: Multi-level credentialization: - **User Level**: Firebase Auth JWT for UI → Backend requests - **Device Level**: Automatic device_id from Bridge's hardware fingerprinting - **Service Level**: Backend uses service account credentials for Forge → GCS/Firestore - **Data Isolation**: User can only analyze own device data

---

🔐 Device Identity Integration

Bridge Device Identity Mechanism (from `bridge/device/identity.py`)

```python
from bridge.device import DeviceIdentity

# Exact same method as Bridge uses:
device_id = DeviceIdentity.get_device_id()
# Returns: "{hostname}-{hardware_fingerprint}"
# Example: "johns-macbook-pro-abc12345"
```

Hardware Fingerprinting Components: - **Windows**: Machine GUID from registry - **macOS**: Hardware serial number via ioreg - **Linux**: /etc/machine-id - **Cross-platform**: hostname, platform, OS user - **Result**: Persistent device_id that survives app reinstalls

Desktop Service Device Resolution

Desktop Service will use **identical device identity logic** as Bridge:

```python
# File: bridge/ui/desktop/device_resolver.py
from bridge.device import DeviceIdentity

class DesktopDeviceResolver:
    @staticmethod
    def get_current_device_id() -> str:
        """Get current device ID using Bridge's method."""
        return DeviceIdentity.get_device_id()

    @staticmethod
    def get_device_identity() -> Dict[str, Any]:
        """Get full device identity with metadata."""
        return DeviceIdentity.get_full_identity()
```

🧪 Comprehensive Credentialization Testing Strategy

Device Identity Testing

```python
# File: tests/integration/test_device_identity_integration.py
class TestDeviceIdentityIntegration:
    def test_desktop_service_uses_bridge_device_identity(self):
        """Verify desktop service gets same device_id as Bridge."""
        bridge_device_id = DeviceIdentity.get_device_id()
        desktop_device_id = DesktopDeviceResolver.get_current_device_id()
        assert bridge_device_id == desktop_device_id

    def test_device_identity_persistence(self):
        """Verify device_id is consistent across sessions."""
        device_id_1 = DeviceIdentity.get_device_id()
        # Clear cache
        DeviceIdentity._cached_identity = None
        device_id_2 = DeviceIdentity.get_device_id()
        assert device_id_1 == device_id_2

    def test_hardware_fingerprint_uniqueness(self):
```

```
        """Verify hardware fingerprinting produces unique IDs."""
        device_id = DeviceIdentity.get_device_id()
        identity = DeviceIdentity.get_full_identity()

        assert len(device_id) > 0
        assert "-" in device_id  # hostname-fingerprint format
        assert identity["hardware_fingerprint"] in device_id
```

User Authorization Testing

```
# File: tests/integration/test_user_authorization.py
class TestUserDeviceAuthorization:
    def test_user_can_only_analyze_own_device(self):
        """User can only request analysis of their current device."""
        user_jwt = create_test_jwt("john@diligent4.com", "diligent4")
        current_device = DeviceIdentity.get_device_id()

        # Valid: same device
        response = client.post("/api/v1/sessions/process",
            headers={"Authorization": f"Bearer {user_jwt}"},
            json={"start_time": "2024-01-15T09:00:00Z", "end_time": "2024-01-15T17:00:00Z"})
        assert response.status_code == 200

        # Invalid: attempting different device (should be impossible in real flow)
        # This test ensures backend rejects any tampering attempts

    def test_cross_org_data_isolation(self):
        """Users cannot access data from other organizations."""
        user_org1 = create_test_jwt("alice@org1.com", "org1")
        user_org2 = create_test_jwt("bob@org2.com", "org2")

        # Each user can only see their own org's data
        response1 = client.get(f"/api/v1/sessions/{session_id}/workflow",
                              headers={"Authorization": f"Bearer {user_org1}"})
        response2 = client.get(f"/api/v1/sessions/{session_id}/workflow",
                              headers={"Authorization": f"Bearer {user_org2}"})

        # Only org1 user should succeed if session belongs to org1
```

End-to-End Credentialization Flow Testing

```
# File: tests/integration/test_e2e_credentialization.py
class TestE2ECredentializationFlow:
    def test_complete_user_session_workflow(self):
        """Test complete flow: User → Bridge → Forge → Firestore → User."""

        # 1. User logs in and gets JWT
        user_jwt = firebase_auth.login("john@diligent4.com")
        device_id = DeviceIdentity.get_device_id()

        # 2. Start session on Bridge
        bridge_response = bridge_client.start_session(
            device_id=device_id,
            collectors=["mouse", "window"],
            auth_token=user_jwt
        )
        session_id = bridge_response["session_id"]

        # 3. Stop session and trigger processing
```

```python
        bridge_client.stop_session(session_id, auth_token=user_jwt)
        backend_response = backend_client.process_session(
            session_id=session_id,
            start_time="2024-01-15T09:00:00Z",
            end_time="2024-01-15T17:00:00Z",
            auth_token=user_jwt
        )

        # 4. Wait for Forge processing to complete
        self.wait_for_processing_completion(session_id, user_jwt)

        # 5. Verify user can retrieve their workflow
        workflow_response = backend_client.get_workflow(
            session_id=session_id,
            auth_token=user_jwt
        )

        assert workflow_response["org_id"] == "diligent4"
        assert workflow_response["device_id"] == device_id
        assert workflow_response["analyzed_by_user"] == "john@diligent4.com"
        assert "workflow_data" in workflow_response

    def test_forge_cli_integration_with_credentials(self):
        """Test that backend correctly calls Forge CLI with right parameters."""
        user_jwt = create_test_jwt("alice@diligent4.com", "diligent4")
        device_id = "test-device-123"

        with patch('subprocess.run') as mock_forge:
            backend_client.process_session(
                session_id="test_session",
                start_time="2024-01-15T09:00:00Z",
                end_time="2024-01-15T17:00:00Z",
                auth_token=user_jwt
            )

            # Verify Forge called with correct parameters
            mock_forge.assert_called_with([
                "forge", "analyze",
                "--org", "diligent4",
                "--device", device_id,
                "--from", "2024-01-15T09:00:00Z",
                "--to", "2024-01-15T17:00:00Z"
            ])
```

Security Testing

```python
# File: tests/security/test_credentialization_security.py
class TestCredentializationSecurity:
    def test_jwt_tampering_prevention(self):
        """Verify system rejects tampered JWT tokens."""
        valid_jwt = create_test_jwt("alice@diligent4.com", "diligent4")
        tampered_jwt = valid_jwt[:-10] + "tampered123"

        response = client.post("/api/v1/sessions/process",
            headers={"Authorization": f"Bearer {tampered_jwt}"},
            json={"start_time": "...", "end_time": "..."})

        assert response.status_code == 401
        assert "invalid token" in response.json()["detail"].lower()
```

```python
    def test_org_isolation_enforcement(self):
        """Verify strict org-level data isolation."""
        # Create sessions for different orgs
        session_org1 = create_session("user1@org1.com", "org1", "device1")
        session_org2 = create_session("user2@org2.com", "org2", "device2")

        # User from org1 tries to access org2 data
        user_org1_jwt = create_test_jwt("user1@org1.com", "org1")
        response = client.get(f"/api/v1/sessions/{session_org2}/workflow",
                              headers={"Authorization": f"Bearer {user_org1_jwt}"})

        assert response.status_code == 403
        assert "access denied" in response.json()["detail"].lower()

    def test_device_spoofing_prevention(self):
        """Verify system prevents device ID spoofing attacks."""
        # This test ensures that even if someone tries to modify device_id
        # in requests, the backend uses the actual device making the request
        user_jwt = create_test_jwt("alice@diligent4.com", "diligent4")
        real_device = DeviceIdentity.get_device_id()

        # Attempt to specify different device (should be ignored/prevented)
        with patch('DeviceIdentity.get_device_id', return_value=real_device):
            response = client.post("/api/v1/sessions/process",
                headers={"Authorization": f"Bearer {user_jwt}"},
                json={
                    "start_time": "2024-01-15T09:00:00Z",
                    "end_time": "2024-01-15T17:00:00Z",
                    "device_id": "fake-device-123"  # This should be ignored
                })

        # Verify backend used real device, not fake one
        assert response.status_code == 200
        # Backend should have used real_device, not fake-device-123
```

PHASE 1: Bridge Session API Integration

Goal: Extend flingoos-bridge with session-based recording API while maintaining existing functionality

Repository: `flingoos-bridge` **Dependencies**: Existing TriggerManager, collectors architecture

1.1: Session Data Model Implementation

Decision/Challenge: Implementing session-based data model to replace continuous collection with user-controlled recording sessions.

Context: Current bridge runs collectors continuously. New desktop service requires session-based recording where users explicitly start/stop data collection for specific time periods. Need to track session metadata, collector configuration, and session status throughout the lifecycle.

Resolution: Create session data model with:

- `bridge/models/session.py` - Session model with UUID, timestamps, collector config
- Session states: `recording` → `stopped` → `uploading` → `complete`
- Database persistence for session recovery after restarts

- Integration with existing TriggerManager architecture

Testing:

- Unit tests: Session creation, UUID uniqueness, timestamp validation, status transitions
- Integration tests: Session persistence, database recovery, cross-restart session continuity

Impact: Foundation for all session-based functionality. Enables user-controlled recording sessions while maintaining existing collector architecture.

Follow-up: Next step is SessionManager component to control collectors based on session state.

1.2: Session Manager Component Implementation

Decision/Challenge: Building SessionManager to control collector lifecycle based on session state while maintaining existing TriggerManager architecture.

Context: Need component to orchestrate collector start/stop operations per session, maintain session state, and ensure data isolation between concurrent sessions. Must integrate with existing TriggerManager without breaking current collector functionality.

Resolution: Implement SessionManager with:

- `bridge/services/session_manager.py` - Core session control logic
- Session-based collector enable/disable functionality
- Integration with TriggerManager for event coordination
- Multi-session support with data isolation

Testing:

- Unit tests: Session control logic, collector state management, error handling
- Integration tests: Multi-session isolation, TriggerManager coordination, concurrent session handling

Impact: Enables session-based control of existing collector architecture without disrupting current continuous collection capability.

Follow-up: Next step is database schema for session persistence and recovery.

1.3: Session Database Schema Implementation

Decision/Challenge: Designing database schema for session persistence, recovery, and file tracking while maintaining existing database architecture.

Context: Sessions must survive application restarts, track associated data files, and provide audit trail for session lifecycle. Must extend existing SQLite database without breaking current storage patterns.

Resolution: Extend database with session tables:

- `bridge/storage/session_db.py` - Session database operations
- Session metadata table: UUID, timestamps, status, collector config
- Session-file mapping table: associate uploaded files with sessions
- Database migration for seamless updates

Testing:

- Unit tests: Schema creation, CRUD operations, constraint validation, migration scripts
- Integration tests: Performance under load, data consistency, recovery procedures

Impact: Provides persistent session tracking and enables session recovery after application restarts.

Follow-up: Next step is Bridge REST API implementation.

1.4: Bridge REST API Framework Setup

- **Deliverable**: FastAPI integration with existing bridge
- **Location**: `bridge/api/server.py`
- **Features**:
  - FastAPI server alongside existing CLI
  - Port configuration
  - Graceful shutdown handling
  - Health check endpoints

Unit Tests:

- API server startup/shutdown
- Port binding
- Health check responses
- Error handling

Integration Tests:

- API server + CLI coexistence
- Resource sharing between components
- Performance impact on existing functionality

1.5: Authentication Middleware Implementation

- **Deliverable**: JWT validation for Bridge API
- **Location**: `bridge/api/auth.py`
- **Features**:
  - JWT token validation (reuse existing auth logic)
  - Session-scoped permissions
  - Rate limiting
  - Request logging

Unit Tests:

- JWT validation logic
- Permission checking
- Rate limiting functionality
- Request/response logging

Integration Tests:

- Authentication with existing backend
- Token refresh handling
- Multi-client support

1.6: Session Control Endpoints Implementation

- **Deliverable**: REST endpoints for session management
- **Location**: `bridge/api/routes/sessions.py`
- **Endpoints**:
  - `POST /api/v1/sessions/start`
  - `POST /api/v1/sessions/stop`
  - `GET /api/v1/sessions/{session_id}/status`
  - `GET /api/v1/sessions/{session_id}/files`

Unit Tests:

- Endpoint request/response validation
- Session ID validation
- Error response formatting
- Input sanitization

Integration Tests:

- End-to-end session flow
- Concurrent session handling
- File upload verification
- Session cleanup

Phase 1.3: Integration with Existing Systems

Step 1.3.1: TriggerManager Session Integration

- **Deliverable**: Session-aware event handling
- **Location**: `bridge/collectors/trigger_manager.py` (extend existing)
- **Features**:
  - Session context in events
  - Session-based collector filtering
  - Session activity tracking

Unit Tests:

- Session context propagation
- Event filtering by session
- Session activity metrics

Integration Tests:

- Multi-session event isolation
- Performance impact on existing collectors
- Event history per session

Step 1.3.2: Uploader Session Support

- **Deliverable**: Session-based file uploads
- **Location**: `bridge/uploader/uploader.py` (extend existing)
- **Features**:
  - Session metadata in uploads

- Session completion detection
- Upload verification per session

Unit Tests:

- Session metadata inclusion
- Upload completion tracking
- Session file grouping

Integration Tests:

- Real GCP uploads with session data
- Upload verification workflow
- Session completion notification

Phase 1 Testing & Validation

Phase 1 Integration Test Suite

Location: `tests/integration/test_session_api.py`

Test Scenarios:

- Complete session workflow (start → collect → stop → upload → verify)
- Multi-session concurrent operation
- Session failure and recovery
- Authentication and authorization
- Performance under load

Phase 1 Acceptance Criteria

- [ ] Sessions can be started/stopped via REST API
- [ ] Session data is isolated between concurrent sessions
- [ ] Session metadata is included in all uploads to GCP
- [ ] Existing CLI functionality remains unaffected
- [ ] All existing tests continue to pass
- [ ] Performance regression < 5%

PHASE 2: Forge Processing Integration

Goal: Extend forge pipeline to accept session-based processing requests

Repository: `flingoos-bridge-backend-service` **Dependencies**: Backend service, Firestore integration

Phase 2.1: Session Processing API

Step 2.1.1: Session Processing Request Model

- **Deliverable**: Request models for session processing
- **Location**: `services/backend/models/session_requests.py`

- **Features**:
    - Session processing request validation
    - Session time range specification
    - Organization context handling
    - Job ID generation

Unit Tests:

- Request model validation
- Time range validation
- Organization access control
- Job ID uniqueness

Step 2.1.2: Forge CLI Integration Endpoint

- **Deliverable**: Backend service endpoint to trigger local Forge CLI processing
- **Location**: `services/backend/routes/session_routes.py` (new)
- **New Endpoints**:
    - `POST /api/v1/sessions/{session_id}/process` - Trigger forge CLI execution
    - `GET /api/v1/sessions/{session_id}/status` - Check processing status
- **Implementation**: Backend service calls local `forge analyze` command with session parameters

Unit Tests:

- Endpoint request validation
- Job creation logic
- Status response formatting
- Error handling

Integration Tests:

- Forge CLI command execution
- Session parameter validation
- Process monitoring and status tracking

Phase 2.2: Forge Pipeline Session Support

Step 2.2.1: Session Data Fetcher

- **Deliverable**: Component to fetch session-specific data from GCP
- **Location**: `services/backend/forge/session_data_fetcher.py`
- **Features**:
    - Time-range based GCP file filtering
    - Session metadata validation
    - Data completeness verification
    - Multi-file type handling (mouse, window, audio, etc.)

Unit Tests:

- Time range filtering logic
- File type detection
- Data validation
- Error handling for missing files

Integration Tests:

- Real GCP bucket access
- Session data retrieval
- Performance with large datasets

Step 2.2.2: Forge CLI Execution Integration

- **Deliverable**: Backend service integration with local Forge CLI tool
- **Location**: Backend service worker process
- **Forge CLI Usage**:
  - **Command**: `forge analyze --org {org_id} --device {device_id} --from {start_time} --to {end_time}`
  - **Output Processing**: Forge processes data locally and uploads to Firestore via `forge upload`
  - **Status Tracking**: Monitor local forge process execution and Firestore upload completion
- **Authentication**: Uses existing GCS credentials and Firestore service account

Unit Tests:

- Session data processing logic
- Workflow generation
- Status update mechanisms
- Output formatting

Integration Tests:

- Complete forge CLI execution workflow
- Local processing with GCS data download
- Firestore upload verification and status tracking

Phase 2.3: Workflow Status Tracking

Step 2.3.1: Firestore Job Status Schema

- **Deliverable**: Job tracking in Firestore
- **Location**: Firestore collections design
- **Collections**:
  - `organizations/{org_id}/processing_jobs/{job_id}`
  - `organizations/{org_id}/session_workflows/{session_id}`
- **Status Values**: `queued`, `processing`, `completed`, `failed`

Unit Tests:

- Schema validation
- Status transition logic
- Data structure tests

Integration Tests:

- Real Firestore operations
- Multi-tenant data isolation
- Concurrent access handling

Step 2.3.2: Firestore Integration for Results Retrieval

- **Deliverable**: Direct Firestore integration for accessing forge-processed workflows
- **Location**: Backend service Firestore client
- **Firestore Collections** (used by Forge CLI):
  - `organizations/{org_id}/workflows/{session_id}/files/` - Workflow files uploaded by forge
  - `organizations/{org_id}/reports/` - Generated reports
- **New Backend Endpoints**:
  - `GET /api/v1/sessions/{session_id}/workflow` - Retrieve workflow from Firestore
  - `GET /api/v1/sessions/{session_id}/status` - Check if forge processing completed

Unit Tests:

- Status retrieval logic
- Error response formatting
- Permission validation

Integration Tests:

- Real-time status updates
- Polling performance
- Multi-user access

Phase 2 Testing & Validation

Phase 2 Integration Test Suite

Location: `tests/integration/test_session_processing.py`

Test Scenarios:

- Complete session processing workflow
- Job status tracking and updates
- Multi-tenant data isolation
- Error handling and recovery
- Performance with realistic datasets

Phase 2 Acceptance Criteria

- [ ] Session processing can be triggered via API
- [ ] Job status is tracked and queryable
- [ ] Processed workflows are stored in Firestore
- [ ] Multi-tenant isolation is maintained
- [ ] Processing handles missing or incomplete data gracefully
- [ ] Existing Forge functionality remains unaffected

PHASE 3: Desktop UI Integration

Goal: Integrate desktop service UI into flingoos-bridge as web-based interface

Repository: `flingoos-bridge` **Dependencies**: Local desktop UI framework, bridge integration

Phase 3.1: Service Architecture

Step 3.1.1: Local Desktop UI Framework Setup

- **Deliverable**: Native desktop UI application
- **Location**: `bridge/ui/desktop/`
- **Features**:
  - **Option 1: Electron + React** - Familiar web technologies, cross-platform
  - **Option 2: Tauri + React** - Rust backend, smaller footprint, better security
  - **Option 3: PyQt/PySide** - Native Python integration, existing ecosystem
  - **Option 4: Tkinter** - Built-in Python, minimal dependencies
- **Integration**: Embedded within bridge.exe executable

Unit Tests:

- Service startup/shutdown
- Configuration loading
- Request routing
- Error handling

Integration Tests:

- Service health checks
- Configuration validation
- Resource cleanup

Step 3.1.2: External Service Clients

- **Deliverable**: HTTP clients for Bridge and Forge APIs
- **Location**: `src/clients/`
- **Features**:
  - Bridge API client with authentication
  - Forge API client with authentication
  - Firestore client for status polling
  - Error handling and retries

Unit Tests:

- Client authentication
- Request/response handling
- Error parsing
- Retry logic

Integration Tests:

- Real API communication
- Authentication token management
- Network error handling

Phase 3.2: Session Orchestration

Step 3.2.1: Session Controller

- **Deliverable**: Core session management logic
- **Location**: `src/controllers/session_controller.py`
- **Features**:
  - Session lifecycle management
  - Bridge API integration
  - Forge API integration
  - Status polling coordination

Unit Tests:

- Session creation and management
- API call orchestration
- Status update handling
- Error recovery logic

Integration Tests:

- End-to-end session workflow
- Multi-session handling
- Error scenarios

Step 3.2.2: Workflow Manager

- **Deliverable**: Workflow retrieval and caching
- **Location**: `src/controllers/workflow_controller.py`
- **Features**:
  - Workflow status polling
  - Workflow data retrieval
  - Local caching for performance
  - Data formatting for UI

Unit Tests:

- Status polling logic
- Workflow retrieval
- Caching mechanisms
- Data transformation

Integration Tests:

- Real Firestore integration
- Polling performance
- Cache consistency

Phase 3.3: Authentication Integration

Step 3.3.1: Auth Service Integration

- **Deliverable**: Authentication with existing systems
- **Location**: `src/auth/`
- **Features**:
  - JWT token management
  - Multi-service authentication
  - Token refresh handling

  - User session management

Unit Tests:

- Token validation and refresh
- Multi-service auth handling
- Session management
- Permission checking

Integration Tests:

- Authentication with Bridge API
- Authentication with Forge API
- Token lifecycle management

Phase 3 Testing & Validation

Phase 3 Integration Test Suite

Location: `tests/integration/test_desktop_service_core.py`

Test Scenarios:

- Complete session orchestration workflow
- Multi-service authentication
- Error handling across services
- Performance under concurrent sessions

Phase 3 Acceptance Criteria

- [ ] Desktop service can orchestrate complete session workflow
- [ ] Authentication works with all external services
- [ ] Sessions are properly isolated and managed
- [ ] Error handling provides meaningful feedback
- [ ] Performance meets requirements (< 2s session start)

PHASE 4: Installer & Distribution Integration

Goal: Integrate desktop service into existing Flingoos Bridge installer and distribution system

Repository: `flingoos-bridge` **Dependencies**: PyInstaller, Inno Setup, auto-updater

4.1: PyInstaller Integration for Desktop UI

Decision/Challenge: Integrating desktop service web UI components and dependencies into existing PyInstaller build system without breaking existing functionality.

Context: Current `flingoos-bridge.spec` builds Windows executable with collectors, tray UI, and updater. Need to add desktop service UI components (likely web-based) while maintaining existing build process and dependencies.

Resolution: Extend PyInstaller configuration:

- Add desktop UI static assets to `datas` collection
- Include web framework dependencies (FastAPI, static files) in `hiddenimports`
- Bundle desktop service templates and assets
- Maintain existing collector and tray functionality

Testing:

- Unit tests: Build system produces working executable with desktop components
- Integration tests: Desktop mode launches without affecting tray mode, resource bundling verification

Impact: Enables single executable distribution with both existing bridge functionality and new desktop service UI.

Follow-up: Next step is Inno Setup installer integration.

4.2: Inno Setup Installer Extension

Decision/Challenge: Extending existing `installer.iss` to support desktop service mode while maintaining current installation and configuration flow.

Context: Current installer prompts for username, sets up auto-start, and configures permissions. Need to add desktop service mode option while maintaining existing workflow for users who prefer tray mode.

Resolution: Enhance installer configuration:

- Add desktop service mode selection in installer UI
- Extend configuration template with desktop service settings
- Add desktop service icons and shortcuts
- Maintain backward compatibility with existing installations

Testing:

- Unit tests: Installer configuration validation, template processing
- Integration tests: Fresh installation with desktop mode, upgrade from existing installation

Impact: Provides unified installation experience with user choice between tray mode and desktop service mode.

Follow-up: Next step is auto-updater integration.

4.3: Auto-Updater Desktop Service Support

Decision/Challenge: Ensuring existing Go auto-updater system properly handles desktop service components during updates.

Context: Current `updater/main.go` handles bridge.exe replacement with UAC elevation, health checks, and restart logic. Desktop service adds web UI components that may require different restart logic.

Resolution: Extend updater functionality:

- Update health check to verify desktop service endpoints
- Modify restart logic to detect desktop vs tray mode
- Ensure UI assets are properly updated

- Maintain existing security and verification systems

Testing:

- Unit tests: Health check logic, restart detection, mode switching
- Integration tests: Complete update workflow in desktop mode, fallback handling

Impact: Ensures seamless auto-updates for desktop service installations without breaking existing tray mode updates.

Follow-up: Next step is version synchronization and build integration.

4.4: Build System Integration

Decision/Challenge: Integrating desktop service components into existing build scripts while maintaining development and CI/CD workflows.

Context: Current `scripts/build-installer.ps1` handles Python dependencies, Go compilation, code signing, and installer generation. Desktop service may add web UI build steps and additional dependencies.

Resolution: Enhance build automation:

- Extend build script with desktop UI compilation steps
- Add desktop service dependencies to requirements
- Update CI/CD workflows to include desktop components
- Maintain existing build verification and testing

Testing:

- Unit tests: Build script execution, dependency resolution
- Integration tests: Complete build pipeline produces working installer

Impact: Provides automated build and distribution pipeline for unified bridge+desktop service application.

Follow-up: Next step is deployment testing and validation.

4.5: Distribution System Validation

Decision/Challenge: Validating complete distribution pipeline ensures desktop service integrates seamlessly with existing deployment infrastructure.

Context: Current distribution uses GitHub Actions, signed releases, and update manifests. Desktop service must integrate without disrupting existing user update workflows or deployment processes.

Resolution: Complete distribution validation:

- Test update manifests include desktop service components
- Verify signature verification works with enhanced executable
- Validate upgrade paths from existing installations
- Ensure deployment scripts handle new components

Testing:

- Unit tests: Update manifest validation, signature verification

- Integration tests: Complete upgrade workflow from current production version

Impact: Ensures reliable distribution of desktop service to existing user base without disruption.

Follow-up: Phase 5 integration testing and production deployment.

PHASE 5: Integration Testing & Deployment

Goal: End-to-end system validation and production deployment preparation

Dependencies: All systems integration

Phase 5.1: System Integration Testing

Step 5.1.1: End-to-End Test Suite

- **Deliverable**: Complete system integration tests
- **Location**: `tests/e2e/`
- **Scenarios**:
  - Complete user workflow (UI → Bridge → GCP → Forge → Firestore → UI)
  - Multi-tenant isolation
  - Concurrent session handling
  - Error recovery workflows
  - Performance under load

Step 5.1.2: Security Testing

- **Deliverable**: Security validation suite
- **Location**: `tests/security/`
- **Focus Areas**:
  - Authentication and authorization
  - Multi-tenant data isolation
  - API security
  - Token handling
  - Data encryption

Phase 5.2: Deployment & Distribution

Step 5.2.1: Deployment Configuration

- **Deliverable**: Production deployment setup
- **Features**:
  - Configuration management
  - Environment separation
  - Monitoring and logging
  - Health checks

Step 5.2.2: Distribution Package

- **Deliverable**: Desktop application installer
- **Features**:
  - Cross-platform installer
  - Auto-update capabilities
  - Configuration wizard
  - Uninstall procedures

Phase 5 Testing & Validation

Phase 5 Acceptance Criteria

- [ ] Complete system works end-to-end
- [ ] Security requirements are met
- [ ] Performance requirements are satisfied
- [ ] Deployment is automated and reliable
- [ ] Distribution package works across target platforms

Success Metrics

Functional Requirements

- [ ] Sessions can be started and stopped from desktop UI
- [ ] Session data is processed automatically after stop
- [ ] Workflows are displayed within 60 seconds of processing completion
- [ ] Multi-tenant data isolation is maintained
- [ ] System handles 10 concurrent sessions per organization

Performance Requirements

- [ ] Session start response time < 2 seconds
- [ ] Workflow processing completion notification < 60 seconds
- [ ] UI remains responsive during all operations
- [ ] System memory usage < 500MB during normal operation

Security Requirements

- [ ] All API communications use JWT authentication
- [ ] Multi-tenant data isolation is verified
- [ ] Audit logs capture all user actions
- [ ] No credentials stored in plaintext

Reliability Requirements

- [ ] System handles network disconnections gracefully
- [ ] Session data is not lost during system failures
- [ ] Error messages are clear and actionable
- [ ] System recovery is automatic where possible

Success Metrics Overview

Phase Dependencies

Phase	Focus	Critical Dependencies
Phase 1	Bridge Session API	TriggerManager, existing collectors
Phase 2	Forge Processing	Backend service, Firestore integration
Phase 3	Desktop UI Integration	Local desktop UI framework, bridge integration
Phase 4	**Installer & Distribution**	**PyInstaller, Inno Setup, auto-updater**
Phase 5	Testing & Deployment	All systems integration

Prerequisites: Access to current build infrastructure, understanding of existing collector architecture

Updated Success Criteria

Phase 4 Critical Success Metrics *(Previously Missing)*:

- [ ] Desktop service installs as single unified application with bridge
- [ ] Existing users can upgrade seamlessly without losing configuration
- [ ] Auto-updater handles desktop service components correctly
- [ ] Installation size increase < 50MB from current bridge installer
- [ ] Build pipeline produces signed, verified installer automatically

---

Implementation Notes

Repository Strategy:

- **Primary Development**: `flingoos-bridge` repository (Phases 1, 3, 4)
- **Backend Integration**: `flingoos-bridge-backend-service` (Phase 2)
- **This Repository**: Planning, documentation, and reference materials only

Development Methodology:

- Phase-based approach with comprehensive testing at each step
- Security validation and integration with existing systems throughout
- Maintains existing battle-tested security and distribution infrastructure

Architecture Benefits:

- ✅ Single unified installation and auto-update system
- ✅ Leverages existing enterprise-grade security (JWT + GCP + Firebase)
- ✅ Maintains backward compatibility with current bridge functionality
- ✅ Reduces maintenance overhead and user complexity

Prerequisites:

- Access to current build infrastructure and development tools
- Understanding of existing collector architecture and TriggerManager system

---

🧪 TESTING STRATEGY: Comprehensive Credentialization Validation

Phase 1 Testing: Bridge Session API + Device Identity

Unit Testing

```python
# File: tests/unit/test_device_identity_desktop.py
class TestDesktopDeviceIdentity:
    def test_desktop_uses_bridge_device_identity(self):
        """Desktop service uses same device ID as Bridge."""
        from bridge.device import DeviceIdentity
        from bridge.ui.desktop.device_resolver import DesktopDeviceResolver

        bridge_id = DeviceIdentity.get_device_id()
        desktop_id = DesktopDeviceResolver.get_current_device_id()

        assert bridge_id == desktop_id
        assert bridge_id.count('-') == 1  # hostname-fingerprint format
        assert len(bridge_id.split('-')[1]) == 8  # 8-char fingerprint

# File: tests/unit/test_session_user_binding.py
class TestSessionUserBinding:
    def test_session_includes_user_context(self):
        """Sessions are created with user context from JWT."""
        user_jwt = create_mock_jwt("alice@org1.com", "org1")
        device_id = DeviceIdentity.get_device_id()

        session = create_session_with_user_context(
            device_id=device_id,
            user_jwt=user_jwt,
            collectors=["mouse", "window"]
        )

        assert session.device_id == device_id
        assert session.user_email == "alice@org1.com"
        assert session.org_id == "org1"
```

Integration Testing

```python
# File: tests/integration/test_bridge_desktop_session_flow.py
class TestBridgeDesktopSessionIntegration:
    def test_user_session_creation_flow(self):
        """Complete user-initiated session creation."""
        # 1. User authenticates in Desktop UI
        user_jwt = firebase_auth.authenticate("alice@org1.com", "password")
        device_id = DeviceIdentity.get_device_id()

        # 2. Desktop UI starts session on Bridge
        response = bridge_client.start_session(
```

```
            device_id=device_id,
            collectors=["mouse", "window", "keyboard"],
            user_context={"jwt": user_jwt, "email": "alice@org1.com"}
        )

        # 3. Verify session metadata
        session_id = response["session_id"]
        session_data = bridge_client.get_session_status(session_id)

        assert session_data["device_id"] == device_id
        assert session_data["user_email"] == "alice@org1.com"
        assert session_data["org_id"] == "org1"
        assert session_data["status"] == "recording"
```

Phase 2 Testing: Forge CLI Integration + Credentialization

Unit Testing

```python
# File: tests/unit/test_forge_cli_integration.py
class TestForgeCLIIntegration:
    def test_forge_command_construction(self):
        """Backend constructs correct Forge CLI command."""
        session_request = {
            "org_id": "org1",
            "device_id": "alice-macbook-abc123",
            "user_email": "alice@org1.com",
            "start_time": "2024-01-15T09:00:00Z",
            "end_time": "2024-01-15T17:00:00Z"
        }

        command = construct_forge_command(session_request)

        expected = [
            "forge", "analyze",
            "--org", "org1",
            "--device", "alice-macbook-abc123",
            "--from", "2024-01-15T09:00:00Z",
            "--to", "2024-01-15T17:00:00Z"
        ]

        assert command == expected

    def test_forge_process_monitoring(self):
        """Backend can monitor Forge process execution."""
        with patch('subprocess.Popen') as mock_process:
            mock_process.return_value.poll.return_value = None  # Running
            mock_process.return_value.returncode = 0  # Success

            forge_executor = ForgeExecutor()
            status = forge_executor.start_processing(session_request)

            assert status.is_running() == True
            mock_process.return_value.poll.return_value = 0  # Finished
            assert status.is_complete() == True
```

Integration Testing

```python
# File: tests/integration/test_user_forge_workflow.py
class TestUserForgeWorkflowIntegration:
    def test_complete_user_to_forge_flow(self):
        """User request → Backend → Forge CLI → Firestore → User result."""

        # 1. Simulate user session request
        user_jwt = create_test_jwt("bob@org2.com", "org2")
        device_id = "bob-laptop-xyz789"

        # 2. Backend processes user request
        with patch('DeviceIdentity.get_device_id', return_value=device_id):
            response = backend_client.process_session(
                session_id="test_session_123",
                start_time="2024-01-15T09:00:00Z",
                end_time="2024-01-15T17:00:00Z",
                auth_token=user_jwt
            )

        assert response.status_code == 202  # Accepted
        job_id = response.json()["job_id"]

        # 3. Wait for processing and verify Firestore upload
        self.wait_for_forge_completion(job_id, timeout=30)

        # 4. Verify Firestore contains user-contextualized data
        firestore_doc = firestore_client.get_workflow("test_session_123")

        assert firestore_doc["org_id"] == "org2"
        assert firestore_doc["device_id"] == device_id
        assert firestore_doc["analyzed_by_user"] == "bob@org2.com"
        assert firestore_doc["status"] == "completed"
        assert "workflow_data" in firestore_doc
```

Phase 3 Testing: Desktop UI Credentialization

Unit Testing

```python
# File: tests/unit/test_desktop_ui_auth.py
class TestDesktopUIAuthentication:
    def test_automatic_device_resolution(self):
        """Desktop UI automatically resolves current device."""
        ui_controller = SessionController()

        # Should automatically detect device without user input
        assert ui_controller.device_id is not None
        assert ui_controller.device_id == DeviceIdentity.get_device_id()
        assert '-' in ui_controller.device_id  # hostname-fingerprint format

    def test_user_session_binding(self):
        """User login binds to current device context."""
        firebase_user = mock_firebase_login("charlie@org3.com")
        ui_controller = SessionController()

        ui_controller.authenticate_user(firebase_user)

        assert ui_controller.user_email == "charlie@org3.com"
        assert ui_controller.org_id == "org3"
        assert ui_controller.device_id == DeviceIdentity.get_device_id()
```

End-to-End Testing

```
# File: tests/e2e/test_complete_credentialization_flow.py
class TestCompleteCredentializationFlow:
    def test_user_complete_session_workflow(self):
        """Complete flow: Login → Record → Process → View Results."""

        # 1. User Authentication
        user_email = "david@org4.com"
        firebase_user = firebase_auth.login(user_email, "test_password")
        desktop_ui = DesktopUI()
        desktop_ui.authenticate(firebase_user)

        # 2. Session Recording
        device_id = desktop_ui.get_current_device_id()
        session_id = desktop_ui.start_recording(["mouse", "window", "audio"])

        # Simulate user activity for 30 seconds
        time.sleep(30)

        desktop_ui.stop_recording(session_id)

        # 3. Trigger Processing
        processing_job = desktop_ui.process_session(
            session_id=session_id,
            time_range=("2024-01-15T09:00:00Z", "2024-01-15T09:01:00Z")
        )

        # 4. Wait for completion and retrieve results
        desktop_ui.wait_for_processing_completion(processing_job.job_id)
        workflow = desktop_ui.get_session_workflow(session_id)

        # 5. Verify complete credentialization chain
        assert workflow["org_id"] == "org4"
        assert workflow["device_id"] == device_id
        assert workflow["analyzed_by_user"] == user_email
        assert workflow["requested_from_device"] == device_id  # Same device
        assert len(workflow["workflow_data"]["segments"]) > 0

        # 6. Verify user can only see their own data
        other_user_jwt = create_test_jwt("eve@org5.com", "org5")
        with pytest.raises(PermissionError):
            desktop_ui.get_session_workflow(session_id, auth_token=other_user_jwt)
```

Security Testing: Credentialization Validation

Authentication Security

```
# File: tests/security/test_credentialization_security.py
class TestCredentializationSecurity:
    def test_device_spoofing_prevention(self):
        """System prevents device ID spoofing attacks."""
        real_device = DeviceIdentity.get_device_id()
        user_jwt = create_test_jwt("attacker@org1.com", "org1")

        # Attempt to manipulate device_id in request
        malicious_request = {
            "session_id": "fake_session",
```

```
            "device_id": "malicious-device-999",  # Fake device
            "start_time": "2024-01-15T09:00:00Z",
            "end_time": "2024-01-15T17:00:00Z"
        }

        # Backend should ignore provided device_id and use actual device
        with patch('DeviceIdentity.get_device_id', return_value=real_device):
            response = backend_client.process_session(
                malicious_request, auth_token=user_jwt
            )

        # Verify backend used real device, not fake one
        job_data = get_processing_job_data(response.json()["job_id"])
        assert job_data["device_id"] == real_device
        assert job_data["device_id"] != "malicious-device-999"

    def test_cross_user_data_isolation(self):
        """Users cannot access other users' session data."""
        # Create sessions for different users
        user1_jwt = create_test_jwt("alice@org1.com", "org1")
        user2_jwt = create_test_jwt("bob@org1.com", "org1")  # Same org

        device1 = "alice-device-123"
        device2 = "bob-device-456"

        # User1 creates session
        with patch('DeviceIdentity.get_device_id', return_value=device1):
            session1_id = create_test_session(user1_jwt)

        # User2 tries to access User1's session
        with patch('DeviceIdentity.get_device_id', return_value=device2):
            response = backend_client.get_workflow(
                session_id=session1_id,
                auth_token=user2_jwt
            )

        # Should fail - different user/device
        assert response.status_code == 403
        assert "access denied" in response.json()["detail"].lower()
```
