LIQUID ABT - Complete Development Documentation

Full Conversation & Technical Specifications

Initial Project Brief

Project Overview

🚀 **LIQUID Automated Bitcoin Treasury Software**: Comprehensive Multi-Tenant Bitcoin Treasury SaaS Platform

🎯 **CORE PURPOSE & VISION**: LIQUID Automated Bitcoin Treasury (ABT) is a comprehensive multi-tenant SaaS platform designed to revolutionize how Australian SMEs accumulate and manage Bitcoin as part of their corporate treasury strategy. The platform automates the conversion of business revenue streams into Bitcoin while maintaining full compliance with Australian tax and regulatory requirements.

Vision: Enable 100,000+ Australian businesses to effortlessly build Bitcoin treasuries through automated, compliant, and secure revenue conversion.

🏗️ MULTI-TENANT SAAS ARCHITECTURE

Platform Structure:

- Master System: PostgreSQL-based tenant management with schema isolation
- Tenant Isolation: Each client gets dedicated database schema (tenant_uuid)
- Subdomain Architecture: {client}.liquidtreasury.business for branded access
- Cross-Tenant Security: JWT validation with tenant context verification
- Scalable Infrastructure: AWS deployment not yet ready (but account created) for 100,000+ businesses

Subscription Tiers:

🆓 **STARTER PLAN (FREE)**

- Target: New businesses testing Bitcoin treasury
- Limits: $50K monthly volume
- Features:

  - Dollar Cost Averaging (DCA)
  - Basic revenue conversion (5% max)
  - Basic reporting dashboard
  - Transaction tracking for easy sharing with accountants/ATO
  - Email notifications
  - Standard support

📈 GROWTH PLAN ($24.99/month)

- Target: Growing businesses with consistent revenue
- Limits: 10 integrations, $500K monthly volume, 10 users
- Features:
  - All Starter features
  - Can give LIQUID access to balance sheet for automated/bespoke treasury settings
  - Advanced rebalancing algorithms
  - Cash sweep functionality
  - Liquidity floor management
  - Advanced reporting & analytics
  - SMS/Email/Slack webhook notifications
  - Priority email support

💼 PRO PLAN ($97.99/month)

- Target: Established businesses with complex needs
- Limits: Unlimited integrations, volume, and users
- Features:
  - All Growth features
  - Access to ZeroCap lending product
  - Profit locking strategies
  - Tax loss harvesting automation
  - Automated tax integration and reporting with accounting software
  - Multi-level approval workflows
  - Dedicated accountant portal
  - Custom API access

    - Phone support

🏢 **ENTERPRISE PLAN (Contact Sales)**

- Target: Large corporations with custom requirements
- Features:
    - All Pro features
    - Dedicated account manager
    - Custom integrations development
    - On-premise deployment options
    - Advanced compliance monitoring
    - 99.99% uptime SLA
    - Priority phone support
    - Custom reporting

Fee Structure:

- Free plan: 1.25% fee (0.5% to ZeroCap 0.75% to us) - $0/month
- Growth plan: 0.55% fee (0.5% to ZeroCap, 0.05% to us) - $24.99/month
- Pro plan: 0.5% fee (0.3% to ZeroCap, 0.2% to us) - $97.99/month
- Enterprise plan: 0.2% fee (0.2% to ZeroCap none to us) - custom pricing

---

Initial Technical Response

🎯 Market-Speed Development Strategy

Phase 1: Core Multi-Tenant MVP (Week 1-2)

Why Multi-Tenant from Day 1:

- Each client needs {client}.liquidtreasury.business subdomain
- Schema isolation is core to the business model
- Subscription tiers require tenant-level feature gating
- Can't retrofit multi-tenancy later without complete rebuild

MVP Feature Set:
✅ Multi-tenant architecture with schema isolation

✅ Stripe OAuth + webhook processing
✅ Percentage-based treasury rules (0.1% to 100%)
✅ Self-custody Bitcoin withdrawals
✅ ZeroCap integration for Bitcoin purchases
✅ Real-time dashboard with BTC/AUD tracking
✅ Transaction fee structure (1.25% for free tier)
✅ Basic Australian tax tracking

🏗️ Technical Architecture

Multi-Tenant Database Strategy

```sql
-- Master tenant management
CREATE TABLE tenants (
  id UUID PRIMARY KEY,
  company_name TEXT,
  subdomain TEXT UNIQUE,
  subscription_tier TEXT,
  created_at TIMESTAMP
);

-- Per-tenant schema isolation
CREATE SCHEMA tenant_${uuid};
-- Each tenant gets: users, integrations, rules, transactions, etc.
```

Treasury Rules Engine

- Percentage-based: Convert X% of each payment
- Threshold-based: Convert when balance hits $X
- DCA: Fixed amounts on schedule
- Advanced: Rebalancing, cash sweep, liquidity floors

Exchange Provider Clarification

User Input:

"Keep ZeroCap as the end liquidity provider we'll use when their API is ready in 2 months, but we'll use Kraken's API for now for auto buying and auto withdrawing to cold wallets. I need the code to be designed to be easy to swap ZeroCap in when they're ready.

I also want to test both Stripe & PayPal. And again, there are a ton more integrations we're planning as you know. Will the code be fine to integrate more platforms in seamlessly for our clients to use?"

Technical Solution: Abstraction Architecture

Critical Abstraction Architecture

```typescript
// Exchange Provider Interface (Bitcoin side)
interface ExchangeProvider {
  name: string;
  purchaseBitcoin(amountAUD: number): Promise<BTCPurchase>;
  withdrawToWallet(btcAmount: number, address: string): Promise<WithdrawalResult>;
  getBalance(): Promise<Balance>;
  getCurrentPrice(): Promise<PriceData>;
}

// Payment Processor Interface (Fiat side)
interface PaymentProcessor {
  name: string;
  type: 'stripe' | 'paypal' | 'square' | 'shopify' | 'tyro';
  initiateOAuth(): Promise<OAuthResult>;
  handleWebhook(payload: any): Promise<Transaction>;
  getTransactions(since: Date): Promise<Transaction[]>;
  refund(transactionId: string): Promise<RefundResult>;
}
```

Project Structure

```
/liquid-abt
  /apps
    /web (Next.js main app)
    /tenant (tenant-specific subdomain app)
  /packages
    /database (Prisma schemas, migrations)
    /treasury-engine (conversion rules logic)
    /integrations
      /exchanges
        /kraken (current implementation)
        /zerocap (future implementation stub)
        /interface.ts (ExchangeProvider interface)
      /payments
        /stripe
        /paypal
        /square
        /shopify
        /tyro
        /interface.ts (PaymentProcessor interface)
    /auth (JWT + multi-tenant auth)
    /ui (shared components)
```

Kraken Integration Implementation

typescript

```
// /packages/integrations/exchanges/kraken/index.ts
import { ExchangeProvider } from '../interface';
import KrakenClient from 'kraken-api';

export class KrakenProvider implements ExchangeProvider {
  private client: KrakenClient;

  constructor(apiKey: string, apiSecret: string) {
    this.client = new KrakenClient(apiKey, apiSecret);
  }

  async purchaseBitcoin(amountAUD: number): Promise<BTCPurchase> {
    // Convert AUD to BTC at current rate
    const ticker = await this.client.api('Ticker', { pair: 'XBTAUD' });
    const btcAmount = amountAUD / ticker.result.XBTAUD.a[0];

    // Place market order
    const order = await this.client.api('AddOrder', {
      pair: 'XBTAUD',
      type: 'buy',
      ordertype: 'market',
      volume: btcAmount
    });

    return {
      btcAmount,
      audSpent: amountAUD,
      exchangeRate: ticker.result.XBTAUD.a[0],
      orderId: order.result.txid[0]
    };
  }

  async withdrawToWallet(btcAmount: number, address: string): Promise<WithdrawalResult> {
    // Validate Bitcoin address first
    if (!this.validateBitcoinAddress(address)) {
      throw new Error('Invalid Bitcoin address');
    }

    // Initiate withdrawal to cold wallet
    const withdrawal = await this.client.api('Withdraw', {
      asset: 'XBT',
      key: address, // Pre-configured withdrawal address
      amount: btcAmount
    });

    return {
      txid: withdrawal.result.txid,
      status: 'pending'
    };
  }
}
```

Exchange Provider Factory

typescript

```typescript
// /packages/integrations/exchanges/index.ts
export class ExchangeFactory {
  static create(provider: string, credentials: any): ExchangeProvider {
    switch(provider) {
      case 'kraken':
        return new KrakenProvider(credentials.apiKey, credentials.apiSecret);
      case 'zerocap':
        return new ZeroCapProvider(credentials);
      default:
        throw new Error(`Unknown exchange provider: ${provider}`);
    }
  }
}

// Easy swap in your treasury engine:
const exchange = ExchangeFactory.create(
  process.env.EXCHANGE_PROVIDER || 'kraken',
  credentials
);
```

Additional Payment Methods: Tyro & Banks

User Input:

"For payments, we also want Tyro (Aussie specific), and traditional bank deposits for businesses who still use those."

Expanded Payment Integration Architecture

Updated Payment Processor Types

typescript

```typescript
// Payment processor types now include banks
type PaymentProcessorType =
  | 'stripe'
  | 'paypal'
  | 'square'
  | 'shopify'
  | 'tyro'
  | 'bank_feed'     // Automated bank connections
  | 'bank_manual';  // Manual bank reconciliation

// Extended interface for different integration patterns
interface PaymentProcessor {
  name: string;
  type: PaymentProcessorType;
  integrationMethod: 'webhook' | 'polling' | 'manual';

  // Standard methods
  initiateConnection(): Promise<ConnectionResult>;
  getTransactions(since: Date): Promise<Transaction[]>;

  // Webhook-based processors only
  handleWebhook?(payload: any): Promise<Transaction>;

  // Bank-specific methods
  reconcileTransaction?(transaction: BankTransaction): Promise<Transaction>;
  importStatement?(file: Buffer, format: 'ofx' | 'csv'): Promise<Transaction[]>;
}
```

Tyro Integration (Australian EFTPOS)

typescript

```
// /packages/integrations/payments/tyro/index.ts
import { PaymentProcessor } from '../interface';

export class TyroProcessor implements PaymentProcessor {
  name = 'Tyro';
  type = 'tyro' as const;
  integrationMethod = 'webhook' as const;

  private apiKey: string;
  private merchantId: string;

  async initiateConnection(): Promise<ConnectionResult> {
    // Tyro uses OAuth 2.0 for partner integrations
    return {
      authUrl: `https://api.tyro.com/oauth/authorize?client_id=${process.env.TYRO_CLIENT_ID}`,
      method: 'oauth2'
    };
  }

  async handleWebhook(payload: any): Promise<Transaction> {
    // Tyro webhooks for real-time EFTPOS transactions
    if (payload.event_type === 'payment.success') {
      return {
        id: payload.transaction_id,
        amount: payload.amount / 100, // Tyro uses cents
        currency: 'AUD',
        type: 'eftpos',
        processor: 'tyro',
        metadata: {
          terminal_id: payload.terminal_id,
          card_type: payload.card_type,
          merchant_reference: payload.reference
        }
      };
    }
  }
}
```

Bank Feed Integration (Automated)

typescript

```
// /packages/integrations/payments/bank-feed/index.ts
import { PaymentProcessor } from '../interface';

export class BankFeedProcessor implements PaymentProcessor {
  name = 'Bank Feed';
  type = 'bank_feed' as const;
  integrationMethod = 'polling' as const; // No webhooks, must poll

  private provider: 'basiq' | 'frollo' | 'wpay'; // Australian bank aggregators

  async initiateConnection(): Promise<ConnectionResult> {
    // Most Australian banks work through aggregators like Basiq
    if (this.provider === 'basiq') {
      return {
        authUrl: await this.getBasiqConsentUrl(),
        method: 'bank_consent'
      };
    }
  }

  async getTransactions(since: Date): Promise<Transaction[]> {
    // Poll bank feed API for new deposits
    const response = await fetch(`https://api.basiq.io/transactions`, {
      headers: {
        'Authorization': `Bearer ${this.accessToken}`,
        'X-User-Id': this.userId
      },
      params: {
        filter: `transactionDate.gte.${since.toISOString()}`,
        filter: 'direction.eq.credit', // Only incoming payments
        filter: 'status.eq.posted'
      }
    });

    // Categorize and filter for business revenue
    return response.data
      .filter(tx => this.isBusinessRevenue(tx))
      .map(this.mapBankTransaction);
  }

  private isBusinessRevenue(tx: BankTransaction): boolean {
    // Smart categorization logic
    const revenueIndicators = [
      'invoice', 'payment', 'sales', 'service',
      'consulting', 'subscription', 'membership'
    ];

    const description = tx.description.toLowerCase();

    // Check for revenue indicators
    return revenueIndicators.some(indicator => description.includes(indicator));
  }
}
```

Universal Australian Banking Question

User Input:

"There's a ton of Aussie banks that SMEs use, how can the code be designed to support easy integrations of any Aussie bank that a SME uses?

Also, i want the bank deposits to also be automated for BTC conversions. Does OAuth apply to banks too?

Nothing in our software should be manual except the clients choosing their bespoke treasury rules and wallet address uploads.

Also, have you considered the automated tax tracking, reporting and integrations with accounting software like Xero, Quickbooks and MYOB?"

Universal Australian Banking & Accounting Architecture

Australian Open Banking (CDR) Architecture

typescript

```
```

```typescript
// The KEY: Consumer Data Right (CDR) covers ALL Australian banks!

interface BankConnector {
  provider: 'cdr' | 'basiq' | 'frollo' | 'bankstatements';
  supportedBanks: string[];

  // OAuth-based bank authentication (YES, banks use OAuth!)
  initiateConsent(bankId: string): Promise<OAuthURL>;
  exchangeToken(code: string): Promise<BankConnection>;

  // Automated transaction fetching
  getAccounts(connection: BankConnection): Promise<BankAccount[]>;
  getTransactions(accountId: string, since: Date): Promise<Transaction[]>;

  // Real-time webhooks (where supported)
  registerWebhook?(accountId: string, url: string): Promise<void>;
}
```

Complete Australian Bank Coverage

typescript

```
// /packages/integrations/banking/bank-registry.ts

export const AUSTRALIAN_BANKS = {
  // Big 4
  'CBA': { cdr: true, basiq: true, name: 'Commonwealth Bank' },
  'NAB': { cdr: true, basiq: true, name: 'National Australia Bank' },
  'ANZ': { cdr: true, basiq: true, name: 'ANZ' },
  'WBC': { cdr: true, basiq: true, name: 'Westpac' },

  // Major Regional Banks
  'BEN': { cdr: true, basiq: true, name: 'Bendigo Bank' },
  'BOQ': { cdr: true, basiq: true, name: 'Bank of Queensland' },
  'SUN': { cdr: true, basiq: true, name: 'Suncorp' },
  'AMP': { cdr: true, basiq: true, name: 'AMP Bank' },
  'MAC': { cdr: true, basiq: true, name: 'Macquarie Bank' },

  // Credit Unions & Building Societies
  'GCU': { cdr: true, basiq: false, name: 'Great Southern Bank' },
  'NCU': { cdr: true, basiq: true, name: 'Newcastle Permanent' },
  'HER': { cdr: true, basiq: false, name: 'Heritage Bank' },
  'PEO': { cdr: true, basiq: true, name: 'People\'s Choice' },
  'BCU': { cdr: true, basiq: false, name: 'Beyond Bank' },

  // Business Banks
  'JUD': { cdr: false, basiq: true, name: 'Judo Bank' },
  'TYR': { cdr: false, basiq: true, name: 'Tyro Bank' },

  // Neo Banks
  'VOLT': { cdr: true, basiq: true, name: 'Volt Bank' },
  'UP': { cdr: true, basiq: true, name: 'Up Bank' },

  // ANY BANK fallback
  'CUSTOM': { cdr: false, basiq: false, bankstatements: true }
};

export class UniversalBankConnector {
  async connectBank(bankCode: string): Promise<BankConnection> {
    const bank = AUSTRALIAN_BANKS[bankCode];

    // Priority order for connection methods
    if (bank.cdr) {
      return this.connectViaCDR(bankCode);
    } else if (bank.basiq) {
      return this.connectViaBasiq(bankCode);
    } else {
      // Fallback: Bank Statements API (works with ANY bank globally)
      return this.connectViaBankStatements(bankCode);
    }
  }
}
```

Consumer Data Right (CDR) Integration

typescript

```typescript
```

```
// /packages/integrations/banking/cdr/index.ts
// CDR is Australia's Open Banking standard - covers ALL accredited banks

export class CDRConnector implements BankConnector {
  provider = 'cdr' as const;

  async initiateConsent(bankId: string): Promise<OAuthURL> {
    // CDR uses OAuth 2.0 with FAPI compliance
    const consent = await fetch('https://api.cdr.gov.au/consent', {
      method: 'POST',
      body: JSON.stringify({
        data_holder: bankId, // Bank's CDR identifier
        permissions: [
          'bank:accounts.basic:read',
          'bank:accounts.detail:read',
          'bank:transactions:read',
          'bank:payees:read',
          'common:customer.basic:read'
        ],
        duration: 31536000 // 1 year consent
      })
    });

    // User redirected to their bank's OAuth page
    return consent.authorization_url;
  }

  async setupRealtimeSync(accountId: string, tenantId: string) {
    // CDR supports webhooks for real-time updates!
    await this.registerWebhook(accountId, `${process.env.API_URL}/webhooks/cdr/${tenantId}`);

    // Now transactions appear in LIQUID instantly when they hit the bank!
  }
}
```

Automated Revenue Detection & Conversion

typescript

```
// /packages/treasury-engine/auto-converter.ts

export class AutomatedTreasuryConverter {
  async processNewBankTransaction(transaction: BankTransaction, tenant: Tenant) {
    // ZERO manual intervention - fully automated flow

    // Step 1: Determine if it's revenue
    const isRevenue = await this.smartRevenueDetection(transaction, tenant);

    if (!isRevenue) {
      return; // Ignore non-revenue transactions automatically
    }

    // Step 2: Apply treasury rules automatically
    const rules = await this.getTreasuryRules(tenant.id);
    const conversionAmount = this.calculateConversion(transaction.amount, rules);

    // Step 3: Execute Bitcoin purchase immediately
    const btcPurchase = await this.purchaseBitcoin(conversionAmount, tenant);

    // Step 4: Withdraw to customer wallet automatically
    await this.withdrawToWallet(btcPurchase.btcAmount, tenant.walletAddress);

    // Step 5: Create accounting entries automatically
    await this.createAccountingEntries(transaction, btcPurchase, tenant);

    // Step 6: Update tax records automatically
    await this.updateTaxRecords(btcPurchase, tenant);

    // DONE! Customer didn't have to do anything!
  }

  private async smartRevenueDetection(tx: BankTransaction, tenant: Tenant): Promise<boolean> {
    // Multi-layer ML detection

    // 1. Basiq/CDR category (if available)
    if (tx.category === 'income' || tx.subCategory === 'business-revenue') {
      return true;
    }

    // 2. Historical pattern matching
    const patterns = await this.getRevenuePatterns(tenant.id);
    if (patterns.matches(tx)) {
      return true;
    }

    // 3. Invoice matching from accounting system
    const matchingInvoice = await this.findInvoiceMatch(tx, tenant);
    if (matchingInvoice) {
      return true;
    }

    // 4. ML confidence scoring
    const confidence = await this.mlRevenueScore(tx);
    if (confidence > 0.95) {
      return true;
    }

    // Only flag for review if confidence is between 0.5-0.95
    if (confidence > 0.5) {
      await this.queueForQuickReview(tx, tenant);
    }

    return false;
  }
}
```

Accounting Software Integration Layer

```
typescript
```

```typescript
// /packages/integrations/accounting/index.ts

interface AccountingProvider {
  name: 'xero' | 'myob' | 'quickbooks';

  // OAuth connection (all use OAuth!)
  connect(): Promise<OAuthURL>;

  // Invoice matching for revenue detection
  getInvoices(since: Date): Promise<Invoice[]>;
  matchTransaction(tx: BankTransaction): Promise<Invoice | null>;

  // Automated journal entries
  createJournalEntry(entry: JournalEntry): Promise<void>;

  // Tax reporting
  getChartOfAccounts(): Promise<Account[]>;
  createTaxReport(period: TaxPeriod): Promise<TaxReport>;

  // Real-time sync
  registerWebhook(events: string[]): Promise<void>;
}
```

Xero Integration (Most Popular in Australia)

typescript

```
// /packages/integrations/accounting/xero/index.ts

export class XeroIntegration implements AccountingProvider {
  async connect(): Promise<OAuthURL> {
    // Xero uses OAuth 2.0
    return {
      url: `https://login.xero.com/identity/connect/authorize?client_id=${this.clientId}&scope=accounting.transactio
      method: 'oauth2'
    };
  }

  async createAutomatedJournalEntry(btcPurchase: BTCPurchase) {
    // Automatically create double-entry bookkeeping
    const journal = {
      date: new Date(),
      reference: `BTC-${btcPurchase.id}`,
      lineItems: [
        {
          account: 'Digital Assets - Bitcoin', // Debit
          description: `Bitcoin purchase via ${btcPurchase.source}`,
          debitAmount: btcPurchase.audAmount
        },
        {
          account: 'Business Bank Account', // Credit
          description: 'Bitcoin treasury allocation',
          creditAmount: btcPurchase.audAmount
        }
      ]
    };

    await this.xeroClient.accounting.createJournal(journal);
  }

  async generateTaxReport(tenant: Tenant, period: TaxPeriod) {
    // Automated CGT calculation
    const btcTransactions = await this.getBTCTransactions(tenant, period);

    const cgtReport = {
      gains: [],
      losses: [],
      method: tenant.cgtMethod || 'FIFO'
    };
    for (const disposal of btcTransactions.disposals) {
      const acquisition = this.matchAcquisition(disposal, tenant.cgtMethod);
      const gainLoss = disposal.audAmount - acquisition.audAmount;

      if (gainLoss > 0) {
        cgtReport.gains.push({
          amount: gainLoss,
          date: disposal.date,
          holdingPeriod: differenceInDays(disposal.date, acquisition.date)
        });
      } else {
        cgtReport.losses.push({
          amount: Math.abs(gainLoss),
          date: disposal.date
        });
      }
    }

    // Create Xero tax report
    await this.createXeroTaxReport(cgtReport);

    return cgtReport;
  }
}
```

Complete Tax Automation

typescript

```
// /packages/tax-engine/australian-tax.ts

export class AustralianTaxAutomation {
  async generateCompleteTaxPackage(tenant: Tenant, financialYear: number) {
    // FULLY AUTOMATED tax reporting

    const reports = {
      // Capital Gains Tax
      cgt: await this.calculateCGT(tenant, financialYear),

      // Business Activity Statement (BAS)
      bas: await this.generateBAS(tenant, financialYear),

      // AUSTRAC reporting (for transactions >$10k)
      austrac: await this.generateAUSTRACReport(tenant, financialYear),

      // Instant Asset Write-Off (if applicable)
      instantWriteOff: await this.checkInstantWriteOff(tenant, financialYear)
    };

    // Push to accounting software automatically
    await this.pushToAccountingSoftware(tenant, reports);

    // Generate ATO-ready file
    const atoFile = await this.generateATOFile(reports);

    return {
      reports,
      atoFile,
      summary: this.generateSummary(reports)
    };
  }
}
```

Advanced Treasury Management Features

User Input:

"Another thing, you'll notice that in the project details, we have tiered subscriptions (Free, Growth, Pro and Enterprise) with more comprehensive features. For the growth, pro and enterprise, we'll give SMEs the option to choose either 'partial' access to business activities, where their live sales can be monitored in real time by our software, and we can perform the tasks they've configured in their settings. But we'll also give them the option to share their full treasury/cash balance/balance sheet for fully automated treasury management/optimisations. For example, they can set their cash balance floor, so that the % of BTC in their treasury never goes above or below a certain amount."

Two-Tier Treasury Access Architecture

Access Level Configuration

```typescript
// /packages/treasury-engine/access-levels.ts

export enum TreasuryAccessLevel {
  PARTIAL = 'partial',  // Sales monitoring only
  FULL = 'full'         // Complete balance sheet access
}

export interface TreasuryConfiguration {
  tenantId: string;
  accessLevel: TreasuryAccessLevel;

  // Partial Access Settings
  partialSettings?: {
    conversionPercentage: number; // 0.1% to 100%
    maxConversionAmount?: number; // Cap per transaction
    minTransactionAmount?: number; // Minimum to trigger
  };

  // Full Access Settings (Growth/Pro/Enterprise only)
  fullAccessSettings?: {
    cashFloor: number;              // Minimum AUD to maintain
    btcAllocationMin: number;       // Min % of treasury in BTC (0-100)
    btcAllocationMax: number;       // Max % of treasury in BTC (0-100)
    targetBtcAllocation: number;    // Target % to maintain
    rebalanceFrequency: 'realtime' | 'hourly' | 'daily';

    // Advanced settings
    upcomingExpensesBuffer: number; // Days of expenses to keep liquid
    seasonalAdjustments: boolean;   // Adjust for seasonal cash flow
    volatilityProtection: boolean;  // Reduce allocation during high volatility
  };
}
```

Balance Sheet Integration Layer

typescript

```
// /packages/integrations/balance-sheet/index.ts

export class BalanceSheetMonitor {
  private accountingConnections: Map<string, AccountingProvider>;

  async getFullFinancialPicture(tenantId: string): Promise<FinancialSnapshot> {
    // Pull from all connected sources
    const [bank, accounting, payments] = await Promise.all([
      this.getBankBalances(tenantId),
      this.getAccountingData(tenantId),
      this.getPaymentProcessorBalances(tenantId)
    ]);

    return {
      // Current State
      totalCash: bank.totalBalance + payments.pendingSettlements,
      bitcoinHoldings: await this.getBitcoinValue(tenantId),
      totalTreasury: bank.totalBalance + payments.pendingSettlements + bitcoinHoldings,

      // Cash Flow Analysis
      averageDailyRevenue: this.calculate30DayAverage(accounting.revenue),
      averageDailyExpenses: this.calculate30DayAverage(accounting.expenses),
      cashBurnRate: this.calculateBurnRate(accounting),
      runwayMonths: this.calculateRunway(bank.totalBalance, this.cashBurnRate),

      // Upcoming Obligations
      upcomingExpenses: await this.getUpcomingExpenses(accounting),
      scheduledPayments: accounting.scheduledPayments,
      taxObligations: accounting.upcomingTax,

      // Historical Patterns
      seasonalFactors: this.analyzeSeasonality(accounting.historicalData),
      growthRate: this.calculateGrowthRate(accounting.revenue)
    };
  }
}
```

Smart Treasury Optimization Engine

typescript

```
// /packages/treasury-engine/optimizer.ts

export class TreasuryOptimizer {
 async optimizeTreasury(tenant: Tenant, snapshot: FinancialSnapshot) {
  const config = tenant.treasuryConfig;

  if (config.accessLevel === TreasuryAccessLevel.PARTIAL) {
   // Simple percentage-based conversion
   return this.handlePartialAccess(tenant, snapshot);
  }

  // FULL ACCESS: Sophisticated optimization
  return this.handleFullAccess(tenant, snapshot);
 }

 private async handleFullAccess(tenant: Tenant, snapshot: FinancialSnapshot) {
  const settings = tenant.treasuryConfig.fullAccessSettings;

  // Step 1: Check cash floor
  const action = this.determineAction(snapshot, settings);

  switch(action.type) {
   case 'EMERGENCY_SELL':
    // Cash below critical floor - sell BTC immediately
    return this.emergencySellBitcoin(action.amount, tenant);

   case 'REBALANCE_SELL':
    // BTC allocation too high - rebalance
    return this.rebalanceSell(action.amount, tenant);

   case 'REBALANCE_BUY':
    // BTC allocation too low - buy more
    return this.rebalanceBuy(action.amount, tenant);

   case 'OPPORTUNISTIC_BUY':
    // Excess cash available - sweep to BTC
    return this.sweepExcessCash(action.amount, tenant);

   case 'HOLD':
    // Everything within targets
    return null;
  }
 }
```

```
private determineAction(snapshot: FinancialSnapshot, settings: FullAccessSettings): TreasuryAction {
 const currentBtcPercent = (snapshot.bitcoinHoldings / snapshot.totalTreasury) * 100;
 const availableCash = snapshot.totalCash;

 // PRIORITY 1: Maintain cash floor
 if (availableCash < settings.cashFloor) {
  const shortfall = settings.cashFloor - availableCash;
  return {
   type: 'EMERGENCY_SELL',
   amount: shortfall * 1.1, // Sell 10% extra for buffer
   reason: 'Cash below floor - emergency liquidity needed'
  };
 }

 // PRIORITY 2: Check upcoming expenses
 const upcomingNeeds = snapshot.upcomingExpenses.next30Days;
 const requiredCash = settings.cashFloor + upcomingNeeds;

 if (availableCash < requiredCash) {
  const shortfall = requiredCash - availableCash;
  return {
   type: 'REBALANCE_SELL',
   amount: shortfall,
   reason: 'Upcoming expenses require additional liquidity'
  };
 }

 // PRIORITY 3: Check BTC allocation limits
 if (currentBtcPercent > settings.btcAllocationMax) {
  // Too much in BTC - sell some
  const targetPercent = settings.targetBtcAllocation;
  const excessBtc = this.calculateExcessBitcoin(snapshot, targetPercent);
  return {
   type: 'REBALANCE_SELL',
   amount: excessBtc,
   reason: `BTC allocation ${currentBtcPercent}% exceeds max ${settings.btcAllocationMax}%`
  };
 }

 if (currentBtcPercent < settings.btcAllocationMin) {
  // Too little in BTC - buy more
  const targetPercent = settings.targetBtcAllocation;
  const neededBtc = this.calculateNeededBitcoin(snapshot, targetPercent);
```

```
      // Only buy if we'll still maintain cash floor
      const cashAfterPurchase = availableCash - neededBtc;
      if (cashAfterPurchase >= requiredCash) {
        return {
          type: 'REBALANCE_BUY',
          amount: neededBtc,
          reason: `BTC allocation ${currentBtcPercent}% below min ${settings.btcAllocationMin}%`
        };
      }
    }

    // PRIORITY 4: Opportunistic buying (excess cash)
    const excessCash = availableCash - (requiredCash * 1.5); // Keep 50% buffer
    if (excessCash > 1000 && currentBtcPercent < settings.targetBtcAllocation) {
      return {
        type: 'OPPORTUNISTIC_BUY',
        amount: excessCash * 0.5, // Buy with half of excess
        reason: 'Excess cash available for Bitcoin accumulation'
      };
    }

    return { type: 'HOLD', reason: 'All parameters within target ranges' };
  }
}
```

Automated Rebalancing Service

typescript

```
// /packages/services/rebalancer.ts

export class AutomatedRebalancer {
 async initialize(tenant: Tenant) {
  const config = tenant.treasuryConfig.fullAccessSettings;

  if (config.rebalanceFrequency === 'realtime') {
   // Set up webhooks for instant rebalancing
   await this.setupRealtimeMonitoring(tenant);
  } else {
   // Schedule periodic checks
   this.scheduleRebalancing(tenant, config.rebalanceFrequency);
  }
 }

 private async setupRealtimeMonitoring(tenant: Tenant) {
  // Monitor all financial events in real-time

  // 1. Bank account changes
  await this.bankWebhook.on('balance_change', async (event) => {
   await this.checkAndRebalance(tenant);
  });

  // 2. Payment processor settlements
  await this.paymentWebhook.on('settlement', async (event) => {
   await this.checkAndRebalance(tenant);
  });

  // 3. Bitcoin price movements (>5%)
  await this.priceMonitor.on('significant_move', async (event) => {
   await this.checkAndRebalance(tenant);
  });

  // 4. Accounting events (new bills, invoices paid)
  await this.accountingWebhook.on('transaction', async (event) => {
   await this.checkAndRebalance(tenant);
  });
 }

 private async checkAndRebalance(tenant: Tenant) {
  // Get current financial picture
  const snapshot = await this.balanceSheetMonitor.getFullFinancialPicture(tenant.id);
```

```
    // Run optimization
    const action = await this.optimizer.optimizeTreasury(tenant, snapshot);

    if (action) {
      // Execute the action
      await this.executeAction(action, tenant);

      // Log for audit trail
      await this.logTreasuryAction(action, tenant);

      // Notify if significant
      if (action.type === 'EMERGENCY_SELL') {
        await this.notifyUrgent(tenant, action);
      }
    }
  }
}
```

Implementation Strategy Discussion

User Question:

"Ok, this is a lot of content we've just gone through. how should i best share this with Claude Code?"

Recommended Phased Approach

The Phased Approach for Claude Code

Phase 1: Core Foundation (Start Here) Give Claude Code this FIRST - it's your MVP:

```
Build LIQUID ABT - a multi-tenant Bitcoin treasury platform for Australian SMEs.

CORE ARCHITECTURE:
1. Multi-tenant PostgreSQL with schema isolation
2. Each tenant gets subdomain: {client}.liquidtreasury.business
3. JWT auth with tenant context

MVP FEATURES (Build these first):
1. Tenant registration and onboarding
2. Stripe Connect OAuth integration
3. Simple percentage-based treasury rules (convert X% of payments)
4. Kraken API for Bitcoin purchases (abstract it so we can swap to ZeroCap later)
5. Self-custody wallet management (customer provides Bitcoin address)
6. Basic dashboard showing transactions and Bitcoin holdings
7. Transaction fee calculation (1.25% for free tier)

Tech stack: Next.js 14, TypeScript, PostgreSQL, Tailwind CSS

Start by creating the database schema and multi-tenant architecture. Then add Stripe integration. Get this working end-to-end first - real payment triggers real Bitcoin purchase.
```

Phase 2: Additional Payment Methods Once Phase 1 is working, add this:

```
Add more payment processors to LIQUID ABT:

1. PayPal Commerce Platform (OAuth + webhooks)
2. Tyro (Australian EFTPOS - OAuth + webhooks)
3. Bank feed integration via Basiq API (covers 160+ Australian banks)

IMPORTANT: Create a PaymentProcessor interface that all integrations implement:
- initiateConnection()
- handleWebhook()
- getTransactions()

This ensures we can easily add Square, Shopify, and others later.

For banks: Use Basiq API with OAuth to automatically pull transactions. Implement smart revenue detection to identify which deposits are business revenue.
```

Phase 3: Full Treasury Management After payments are working, add the sophisticated features:

```
Implement two-tier treasury access:

PARTIAL ACCESS (all plans):
- Just converts percentage of sales to Bitcoin
- What we already built

FULL ACCESS (Growth/Pro/Enterprise only):
- Integrate with Xero/MYOB/QuickBooks via OAuth
- Pull complete balance sheet data
- Implement cash floor management:
  * If cash < $X, automatically sell Bitcoin
  * If Bitcoin allocation > Y%, rebalance
  * If excess cash > $Z, sweep to Bitcoin
- Real-time monitoring and automatic rebalancing

Create a TreasuryOptimizer class with this priority:
1. Maintain minimum cash (never go below floor)
2. Cover upcoming expenses
3. Stay within BTC allocation limits (min/max %)
4. Opportunistic buying with excess cash
```

Phase 4: Tax & Compliance Once treasury management works:

```
Add Australian tax automation:

1. Connect to Xero/MYOB/QuickBooks
2. Automatic journal entries for every Bitcoin transaction
3. CGT calculation (FIFO, LIFO, Weighted Average)
4. Generate ATO-compliant reports
5. BAS preparation
6. AUSTRAC compliance for transactions >$10k

Everything should be automated - no manual entries needed.
```

The Smart Feeding Strategy

Option A: Incremental Building (Recommended)

1. Copy Phase 1 to Claude Code
2. Let it build and test the MVP
3. Once working, feed Phase 2

4. Continue with Phases 3 & 4

Advantages:

- You get working software quickly
- Can test with real users early
- Easier to debug issues
- Claude Code won't get overwhelmed

Option B: Full Context Dump (If you want everything at once) Create a single document with this structure:

- PROJECT: LIQUID ABT - Automated Bitcoin Treasury for Australian SMEs
- OVERVIEW: [Executive summary]
- PHASE 1 - MVP: [Core features to build first]
- PHASE 2 - PAYMENT INTEGRATIONS: [All payment processor details]
- PHASE 3 - TREASURY MANAGEMENT: [Full access features]
- PHASE 4 - TAX AUTOMATION: [Compliance features]
- TECHNICAL ARCHITECTURE: [Database schemas, API structures, abstraction patterns]
- BUILD ORDER: Start with Phase 1 MVP, get it working end-to-end, then add Phase 2, 3, 4 incrementally

---

Key Implementation Insights

The Competitive Moat

This creates an **INSANE** competitive advantage:

1. **Partial Access** = What competitors offer (basic conversion)
2. **Full Access** = What Fortune 500 companies pay millions for

You're democratizing institutional-grade treasury management for SMEs at $24.99-$97.99/month!

The Key Differentiators

1. **Multi-Tenant Architecture from Day 1**
    - Each client gets {client}.liquidtreasury.business subdomain
    - Schema isolation is core to the business model

- Can't retrofit multi-tenancy later without complete rebuild

2. **Universal Payment & Banking Coverage**
   - Support ALL Australian banks via CDR (Open Banking) OAuth
   - Fallback to Basiq API (covers 160+ banks) with OAuth
   - Ultimate fallback to Bank Statements API (works with ANY bank globally)
   - ALL banks use OAuth-style authentication (no manual imports)
3. **Complete Automation**
   - No manual transaction categorization (ML does it)
   - No manual bank imports (OAuth connections)
   - No manual accounting entries (automated via API)
   - No manual tax calculations (computed in real-time)
   - Customer only sets treasury rules once, everything else is automatic
4. **Sophisticated Treasury Management**
   - Never lets the business run out of cash
   - Maximizes Bitcoin accumulation safely
   - Adjusts to real business needs dynamically
   - Requires ZERO ongoing management

Final Implementation Note

The system should be so automated that a business owner can:

1. Connect their bank and accounting software (5 minutes)
2. Set treasury rules (2 minutes)
3. Add Bitcoin wallet address (1 minute)
4. Never touch the system again - it just works!

Build the bank connection layer to easily add new banks by just adding their CDR/Basiq identifier. Make the accounting layer modular so adding new accounting packages is just implementing an interface.

With proper abstraction layers, adding new payment processors is just implementing an interface. Swapping from Kraken to ZeroCap is literally changing one environment variable. This architecture scales to 50+ payment integrations without any refactoring!

Banking API Provider Flexibility - Nuvei & Monoova

Additional Context

While awaiting response from Basiq regarding their open banking API, calls are scheduled with Nuvei (https://www.nuvei.com/) and Monoova (https://www.monoova.com/) for their open banking APIs. The architecture is designed to easily plug either of these businesses into the workflow.

How Nuvei/Monoova Fit Into The Architecture

Current Banking Provider Interface

typescript

```typescript
interface BankConnector {
  provider: 'cdr' | 'basiq' | 'frollo' | 'bankstatements' | 'nuvei' | 'monoova';
  supportedBanks: string[];

  // OAuth-based bank authentication
  initiateConsent(bankId: string): Promise<OAuthURL>;
  exchangeToken(code: string): Promise<BankConnection>;

  // Automated transaction fetching
  getAccounts(connection: BankConnection): Promise<BankAccount[]>;
  getTransactions(accountId: string, since: Date): Promise<Transaction[]>;

  // Real-time webhooks
  registerWebhook?(accountId: string, url: string): Promise<void>;
}
```

Adding Nuvei Integration

typescript

```typescript
// /packages/integrations/banking/nuvei/index.ts

export class NuveiConnector implements BankConnector {
  provider = 'nuvei' as const;

  async initiateConsent(bankId: string): Promise<OAuthURL> {
    // Nuvei's open banking OAuth flow
    const consent = await fetch('https://api.nuvei.com/openbanking/consent', {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${this.apiKey}`,
      },
      body: JSON.stringify({
        institution_id: bankId,
        permissions: ['accounts', 'transactions', 'balance'],
        redirect_uri: process.env.NUVEI_REDIRECT_URI
      })
    });

    return {
      url: consent.data.auth_url,
      method: 'oauth2'
    };
  }

  async getTransactions(accountId: string, since: Date): Promise<Transaction[]> {
    // Nuvei provides enriched transaction data
    const response = await fetch(`https://api.nuvei.com/accounts/${accountId}/transactions`, {
      headers: { 'Authorization': `Bearer ${this.accessToken}` },
      params: {
        from_date: since.toISOString(),
        to_date: new Date().toISOString()
      }
    });

    // Nuvei likely provides merchant categorization
    return response.data.map(tx => ({
      id: tx.transaction_id,
      amount: tx.amount,
      description: tx.description,
      merchantName: tx.merchant?.name,
      category: tx.category,
      isRevenue: this.determineIfRevenue(tx)
    }));
  }

  async registerWebhook(accountId: string, url: string): Promise<void> {
    // Nuvei real-time transaction notifications
    await fetch('https://api.nuvei.com/webhooks', {
      method: 'POST',
      headers: { 'Authorization': `Bearer ${this.apiKey}` },
      body: JSON.stringify({
        account_id: accountId,
        webhook_url: url,
        events: ['transaction.created', 'transaction.updated']
      })
    });
  }
}
```

Adding Monoova Integration

```
typescript
```

```
// /packages/integrations/banking/monoova/index.ts

export class MonoovaConnector implements BankConnector {
  provider = 'monoova' as const;

  async initiateConsent(bankId: string): Promise<OAuthURL> {
    // Monoova's PayID and NPP integration
    const session = await fetch('https://api.monoova.com/openbanking/auth/initiate', {
      method: 'POST',
      headers: {
        'x-api-key': this.apiKey,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        bank: bankId,
        scope: 'accounts transactions balance',
        callback_url: process.env.MONOOVA_CALLBACK_URL
      })
    });

    return {
      url: session.data.authorization_url,
      method: 'oauth2'
    };
  }

  async getTransactions(accountId: string, since: Date): Promise<Transaction[]> {
    // Monoova specializes in real-time payments (NPP)
    const txns = await fetch(`https://api.monoova.com/v1/accounts/${accountId}/transactions`, {
      headers: {
        'x-api-key': this.apiKey,
        'Authorization': `Bearer ${this.accessToken}`
      },
      params: {
        startDate: since.toISOString(),
        endDate: new Date().toISOString(),
        status: 'Complete'
      }
    });

    // Monoova provides PayID and real-time payment data
    return txns.data.map(tx => ({
      id: tx.transactionId,
      amount: tx.amount,
      description: tx.narrative || tx.description,
      payId: tx.payId, // Unique to Monoova!
      isRealTimePayment: tx.paymentType === 'NPP',
      isRevenue: this.analyzeRevenue(tx)
    }));
  }

  // Monoova speciality: Real-time webhook for instant notifications
  async registerWebhook(accountId: string, url: string): Promise<void> {
    await fetch('https://api.monoova.com/v1/webhooks/subscribe', {
      method: 'POST',
      headers: { 'x-api-key': this.apiKey },
      body: JSON.stringify({
        url: url,
        eventTypes: ['Received_Credits', 'Transaction_Update'],
        accountNumber: accountId
      })
    });
  }
}
```

Updated Universal Bank Connector

```
typescript
```

```
// /packages/integrations/banking/index.ts

export class UniversalBankConnector {
  private providers: Map<string, BankConnector> = new Map();

  constructor() {
    // Initialize all available providers
    this.providers.set('cdr', new CDRConnector());
    this.providers.set('basiq', new BasiqConnector());
    this.providers.set('nuvei', new NuveiConnector());
    this.providers.set('monoova', new MonoovaConnector());
  }

  async connectBank(bankCode: string, preferredProvider?: string): Promise<BankConnection> {
    // Use preferred provider if specified
    if (preferredProvider && this.providers.has(preferredProvider)) {
      const provider = this.providers.get(preferredProvider);
      return provider.initiateConsent(bankCode);
    }

    // Otherwise, use the best available provider for this bank
    const bank = AUSTRALIAN_BANKS[bankCode];

    // Priority order based on capabilities
    if (bank.monoova && this.providers.has('monoova')) {
      // Monoova for real-time NPP payments
      return this.providers.get('monoova').initiateConsent(bankCode);
    } else if (bank.nuvei && this.providers.has('nuvei')) {
      // Nuvei for global reach
      return this.providers.get('nuvei').initiateConsent(bankCode);
    } else if (bank.cdr) {
      // CDR for official Australian open banking
      return this.providers.get('cdr').initiateConsent(bankCode);
    } else if (bank.basiq) {
      // Basiq as fallback
      return this.providers.get('basiq').initiateConsent(bankCode);
    }

    // Ultimate fallback
    return this.providers.get('bankstatements').initiateConsent(bankCode);
  }
}
```

Environment Configuration

```env
# Banking Provider Selection
BANKING_PROVIDER=monoova  # or 'nuvei', 'basiq', 'cdr'

# Monoova Configuration
MONOOVA_API_KEY=xxx
MONOOVA_CALLBACK_URL=https://api.liquidtreasury.business/callbacks/monoova

# Nuvei Configuration
NUVEI_API_KEY=xxx
NUVEI_CLIENT_ID=xxx
NUVEI_REDIRECT_URI=https://api.liquidtreasury.business/callbacks/nuvei

# Basiq Configuration (if you get access later)
BASIQ_API_KEY=xxx
```

Key Advantages of Each Provider

Monoova Advantages:

- **NPP (New Payments Platform)** - Real-time payments, instant settlement
- **PayID Integration** - Easier customer identification
- **Australian-focused** - Deep local bank integrations
- **Webhooks** - Real-time transaction notifications
- **Great for:** Australian SMEs wanting instant Bitcoin conversions

Nuvei Advantages:

- **Global reach** - Works internationally if you expand
- **Payment processing + banking** - Could replace multiple providers
- **Advanced fraud detection** - Built-in security features
- **Multi-currency** - Ready for international expansion
- **Great for:** Scaling beyond Australia

Simple Provider Swap

To switch providers, literally just change one line:

```typescript
// Before (using Basiq)
const bankConnector = new BasiqConnector(config);

// After (using Monoova)
const bankConnector = new MonoovaConnector(config);

// Or even better, use environment variable
const Provider = providers[process.env.BANKING_PROVIDER];
const bankConnector = new Provider(config);
```

What to Ask on Your Calls

For Monoova:

1. "Do you support webhooks for real-time transaction notifications?"
2. "Can we get merchant categorization or transaction enrichment?"
3. "What's your coverage of Australian banks?"
4. "Do you support PayID/NPP for instant settlements?"
5. "What's the pricing model?"

For Nuvei:

1. "Does your open banking API cover Australian banks specifically?"
2. "Can we get real-time webhooks for new transactions?"
3. "Do you provide transaction categorization?"
4. "Can we white-label the consent flow?"
5. "What's the API rate limit?"

The Beauty of Your Architecture

Because you built with the `BankConnector` interface, you can:

- **A/B test providers** - Some tenants use Monoova, others use Nuvei
- **Fallback gracefully** - If one provider is down, switch to another
- **Mix and match** - Use Monoova for Australian banks, Nuvei for international
- **Easy migration** - Switch all tenants from one provider to another with a config change

Your code doesn't care WHICH provider you use - it just calls:

```typescript
const transactions = await bankConnector.getTransactions(accountId, since);
```

This is EXACTLY why the abstraction layer was built - to make provider swapping seamless!

Pro tip: During calls with providers, ask about their **webhook capabilities** for real-time notifications. This is what will make Bitcoin conversions truly instant - the moment money hits the bank account, conversion to Bitcoin can happen without any polling delay!

Summary

LIQUID ABT is positioned to become the dominant Bitcoin treasury automation platform for Australian SMEs by:

1. **Supporting every payment method** - From Stripe to traditional bank deposits
2. **Working with every Australian bank** - Via CDR OAuth connectivity, or Nuvei/Monoova as alternatives
3. **Automating all tax compliance** - Through Xero/MYOB/QuickBooks integration
4. **Providing institutional-grade treasury management** - At SME-friendly prices
5. **Requiring zero manual work** - Complete automation after initial setup
6. **Provider flexibility** - Easy swapping between Basiq, Nuvei, Monoova, or any future banking API provider

The phased implementation approach ensures you can get to market quickly while maintaining the architecture needed for long-term scale to 100,000+ businesses.

This is not just a "Bitcoin buying tool" - it's a complete automated treasury management platform that happens to use Bitcoin as the treasury asset. That positioning allows you to charge premium prices while delivering massive value to Australian SMEs.